AIAA/ANSI
R-013-1992

Recommended Practice

Software Reliability

ANSI/AIAA
R-013-1992

American National Standard

Recommended Practice for Software Reliability

Sponsor

American Institute of Aeronautics and Astronautics

Approved February 23, 1993

American National Standards Institute

Abstract

This recommended practice describes an approach to estimating and predicting the reliability of software. It provides information necessary for the application of software reliability measurement to a project, lays a foundation for building consistent methods, and establishes the basic principle for collecting the performance data needed to assess the reliability of software. The document describes how any user may participate in on-going, software reliability assessments or conduct site or package specific studies.

American National Standard

Approval of an American National Standard requires verification by ANSI that the requirements for due process, consensus, and other criteria have been met by the standards developer.

Consensus is established when, in the judgement of the ANSI Board of Standards Review, substantial agreement has been reached by directly and materially affected interests. Substantial agreement means much more than a simple majority, but not necessarily unanimity. Consensus requires that all views and objections be considered, and that a concerted effort be made toward their resolution.

The use of American National Standards is completely voluntary; their existence does not in any respect preclude anyone, whether he has approved the standards or not, from manufacturing, marketing, purchasing, or using products, processes, or procedures not conforming to the standards.

The American National Standards Institute does not develop standards and will in no circumstances give an interpretation of any American National Standard. Moreover, no person shall have the right or authority to issue an interpretation of an American National Standard in the name of the American National Standards Institute. Requests for interpretations should be addressed to the secretariat or sponsor whose name appears on the title page of this standard.

CAUTION NOTICE: This American National Standard may be revised or withdrawn at any time. The procedures of the American National Standards Institute require that action be taken to affirm, revise, or withdraw this standard no later than five years from the data of approval. Purchasers of American National Standards may receive current information on all standards by calling or writing the American National Standards Institute.

Recommended practice for software reliability / sponsor,
American Institute of Aeronautics and Astronautics ; Space-based Observation Systems Committee on Standards, Software Reliability Working Group.
p. cm.
"R-013-1992."
Includes bibliographical references.
ISBN 1-56347-024-1
1. Computer software--Reliability. I. American Institute of Aeronautics and Astronautics. Space-based Observation Systems Committee on Standards. Software Reliability Working Group.
QA76.76.R44R43 1993
005.1'4--dc20 92-45773
CIP

Published by
American Institute of Aeronautics and Astronautics
370 L'Enfant Promenade, SW, Washington, DC 20024

Printed in the United States of America

CONTENTS

APPENDICES

FOREWORD

This American National Standard Recommended Practice for Software Reliability has been sponsored by the American Institute of Aeronautics and Astronautics (AIAA) as part of its standards program. It originated within the Space-Based Observation Systems Committee on Standards (SBOS/CoS) and was developed by the Software Reliability Working Group. Members of the working group served voluntarily and without compensation; they are not necessarily members of AIAA. This document represents a consensus of opinions on software reliability measurement from individuals inside and outside AIAA who have expressed an interest in participating in the development of the recommended practice.

Software reliability engineering (SRE) is an emerging discipline. This recommended practice describes an approach to estimating and predicting the reliability of software and is intended to provide a foundation on which practitioners and researchers can build consistent methods. It is intended to meet the needs of software practitioners and users who are confronted with varying terminology for reliability measurement and a plethora of models and data collection methods. This recommended practice contains information necessary for the application of software reliability measurement to a project. It includes guidance on the following:

- Common terminology
- Software reliability estimation procedure
- Model selection
- Data collection procedure for use with the AIAA software reliability database
- Open research questions
- Predicting system failure rates.

This recommended practice was developed to meet the needs of software reliability practitioners and researchers. Practitioners are considered to be the following:

- Managers
- Technical managers and acquisition specialists
- Software engineers
- Quality and reliability engineers.

Sections 1-4 should be read by all recommended practice users. Section 5 and Appendices E and F provide the basis for establishing the process and the potential uses of the process. Section 7 provides the foundation for establishing a software reliability data collection program, as well as what information needs to be collected to support the recommended models described in Section 6 and Appendix A. Appendix B identifies tools that support the reliability database, the recommended models and the analysis techniques described in Section 5 and Appendices E and F . Finally, to improve the state of the art in software reliability engineering continuously, Appendix D describes research opportunities for consideration. Recommended Practice users typically review Chapters 1-4 and begin applying the techniques described in Sections 5, 6 and 7, concluding with the appendix on reliability tools.

The AIAA Standards Procedures provide that all approved Standards, Recommended Practices, and Guides are advisory only. Their use by anyone engaged in industry or trade is entirely voluntary. There is no agreement to adhere to any AIAA standards publication and no commitment to conform to or be guided by any standards report. In formulating, revising, and approving standards publications, the Committees on Standards will not consider patents which may apply to the subject matter. Prospective users of the publications are responsible for protecting themselves against liability for infringement of patents, or copyrights, or both.

The viewpoints expressed in this recommended practice are subject to change, depending on developments in the state of the art and comments received from users of the recommended practice. Comments are welcome from any interested party, regardless of membership affiliation with AIAA. Comments should be directed to:

AIAA Headquarters
Standards Department
370 L'Enfant Promenade, SW
Washington, DC 20024-2518

At the time this recommended practice was completed, the Software Reliability Working Group had the following members:

David M. Siefert (NCR Corporation), Working Group Chairman
Ted Keller (IBM Corporation), Vice Chm.
George Stark(Mitre Corporation), Vice Chm. and Tools Team Chair
William Farr (Naval Surface Warfare Cntr), Recommended Models Team Chair
Stephen Kelly (Kaman Sciences), Database Team Chair

Herbert Hecht
Myron Lipow
Michael Lyu
John Musa
Andrea Sebera
Victor Selman
Martin Shooman
Allen Nikora
Norman Schneidewind

Other SBOS/CoS Members who participated in the project:

Frank Ackerman
Myles R. Berg
Charles A. Beswick
Ben Bly
James A. Boyd
Anthony Bukowski
John Collins
Garrett C. Covington
Michael Dewalt
Hartwig Dirscherl
Janet R. Dunham
William Everett
George Finelli
Dean Garlick
Richard Grimaldi
Shahid Habib
David Hamilton
Allen Hankinson
Rick Karcich
Bernice J. Mays
Martha McClure
F. A. Patterson-Hine
J. Raja
George Schick
V. Devon Smith
Richard D. Stutzke
Paul F. Uhlir
Stephanie White
Ken. S. Williamson
Frank Y. Yap

The following individuals have contributed to the review and input of this document:

Bev Littlewood
Harvey Fiala
J. Rayon
Lisa Brownsword
Roger Martin
Robert Tausworthe

Supporting Organizations:

Aerospace
American University
AT&T Corporation
Ball Aerospace
Bendix
Computer Sciences Corporation
Embassy of India
General Dynamics Corporation
Grumman Corporation
Hughes Aircraft
IBM Corporation
John Hopkins Univ. Applied Physics Lab.
Jet Propulsion Laboratory
Kaman Sciences/DACS
Lockheed Company
Loral Space Systems
MITRE Corporation
NASA
National Institute for Standards & Technology
Naval Postgraduate School
Naval Surface Warfare Center
Orien Polytechnic University
Rome Air Develoment Center
Research Triangle Institute
Science Applications International Corp.
SoHaR
Storage Technology Corporation.
TRW
UNISYS
USSC/ANS

The AIAA SBOS/CoS (Andrea F. Sebera, Chair) approved the document in May 1992.

The AIAA Standards Technical Council (William. W. Vaughan, Chairman) approved the document in November 1992.

1.0 INTRODUCTION

1.1 Scope

Software Reliability Engineering (SRE) is an emerging discipline. SRE is the application of statistical techniques to data collected during system development and operation to specify, predict, estimate, and assess the reliability of software-based systems. This recommended practice defines a practical methodology for software reliability engineering.

The Recommended Practice for Software Reliability provides a foundation for practitioners and researchers. It supports the need of software practitioners who are confronted with inconsistent methods and varying terminology for reliability estimation and prediction, as well as a plethora of models and data collection methods. It supports researchers by defining common terms, by identifying criteria for model comparison, and by identifying open research problems in the field.

This document provides guidance on the following:

- Common terminology
- Software reliability estimation and procedure
- Model selection
- Data collection procedure for use with the AIAA software reliability database

This recommended practice is applicable to in-house, commercial, and third-party software projects. It has been developed to support a systems reliability approach. As illustrated in Figure 1, the AIAA Software Reliability Engineering Recommended Practice considers hardware and ultimately systems characteristics.

1.2 Purpose

The AIAA Recommended Practice for Software Reliability is intended to be used from the start of the integration test phase through the operational use phase of the software life cycle. It also provides input to the planning process for reliability management. It is assumed that the use of this handbook has been preceded by an identification and analysis of user requirements.

The Recommended Practice describes activities and qualities of a software reliability estimation and prediction program. It describes a framework that permits assessment of risk and prediction of failure rates, recommends a set of models for software reliability estimation and prediction, and specifies mandatory as well as recommended data collection requirements.

1.3 Intended Audience and Benefits

The Recommended Practice is intended for use by both practitioners (e.g., software developers, software acquisition personnel, technical managers, and quality and reliability personnel) and researchers. Its purpose is to provide both practitioners and researchers with a common baseline for discussion and to define a procedure for assessing the reliability of software. It is assumed that users of this

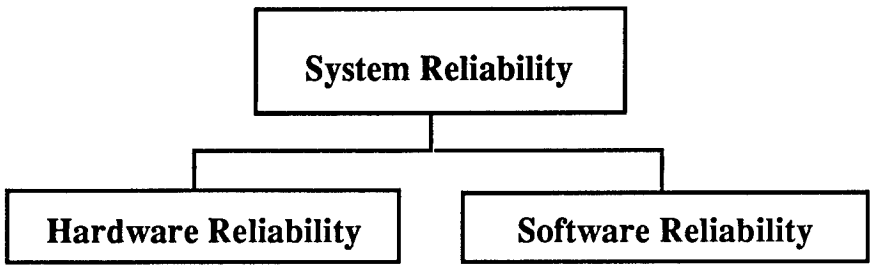

Figure 1 System Reliability Characteristics

recommended practice have a basic understanding of the software life cycle and an understanding of statistical concepts.

This recommended practice is intended to be used in support of designing, developing and testing software. This includes software quality and software reliability activities. It also serves as a reference for research on the subject of software reliability.

1.4 Applications of Software Reliability Engineering

The techniques and methodologies presented in this handbook have been successfully applied to software projects by industry practitioners in order to do the following:

- Determine whether a specific software process is likely to produce code which satisfies a given software reliability requirement,
- Determine the size of a software maintenance effort by predicting the failure rate during the operational phase,
- Provide a metric for process improvement evaluation,
- Assist software safety certification,
- Determine when to release a software system, or to stop testing it,
- Estimate the occurrence of the next failure for a software system,
- Identify elements in a software system which are leading candidates for re-design to improve reliability,
- Measure reliability of a software system in operation, using this information to control change to the system.

It is the intent of this recommended practice to enable other software practitioners to make similar determinations for their particular software systems, as needed. Special attention should be given in the application of these practices to avoid violation of the assumptions inherent in each modeling technique. Data acquisition procedures and model selection criteria are provided and discussed in order to assist in these efforts.

1.5 Relationship to Hardware Reliability

The creation of software and hardware products are alike in many ways and can be similarly managed throughout design and development. While the management techniques may be similar, there are genuine differences between hardware and software [LIPO86, KLIN80]. For example:

- Changes to hardware require a series of important and time-consuming steps: capital equipment acquisition, component procurement, fabrication, assembly, inspection, test and documentation. Changing software is frequently more feasible (although effects of the changes are not always clear) and oftentimes requires only testing and documentation.
- Software has no physical existence. It includes data as well as logic. Any item in a file can be a source of failure.
- Software does not wear out. Furthermore, failures attributable to software faults come without advance warning and often provide no indication they have occurred. Hardware, on the other hand, often provides a period of graceful degradation.
- Software may be more complex than hardware, although exact software copies can be produced, whereas manufacturing limitations affect hardware.
- Repair generally restores hardware to its previous state. Correction of a software fault always changes the software to a new state.
- Redundancy and fault tolerance for hardware are common practice. These concepts are only beginning to be practiced in software.
- Software developments have traditionally made little use of existing components. Hardware is manufactured with standard

parts.

- Hardware reliability is expressed in wall clock time. Software reliability is expressed in execution time.
- A high rate of software change can be detrimental to software reliability.

Despite the above differences, hardware and software reliability must be managed as an integrated system attribute. However, these differences must be acknowledged and accommodated by the techniques applied to each of these two types of subsystems in reliability analyses.

2.0 TERMINOLOGY

This chapter defines terms that are commonly used throughout the recommended practice. The bases for most definitions are from the ANSI / IEEE Standard Glossary of Software Engineering Terminology, STD-729-1991.

Calendar time - Chronological time, including time during which a computer may not be running.

Clock time - Elapsed wall clock time from the start of program execution to the end of program execution.

Error - (1) A discrepancy between a computed, observed or measured value or condition and the true, specified or theoretically correct value or condition. (2) Human action that results in software containing a fault. Examples include omission or misinterpretation of user requirements in a software specification, and incorrect translation or omission of a requirement in the design specification. This is not a preferred usage.

Execution time - (1) The amount of actual or central processor time used in executing a program. (2) The period of time during which a program is executing.

Failure - (1) The inability of a system or system component to perform a required function within specified limits. A failure may be produced when a fault is encountered and a loss of the expected service to the user results. (2) The termination of the ability of a functional unit to perform its required function. (3) A departure of program operation from program requirements.

Failure rate - (1) The ratio of the number of failures of a given category or severity to a given period of time; for example, failures per second of execution time, failures per month. Synonymous with failure intensity. (2) The ratio of the number of failures to a given unit of measure; for example, failures per unit of time, failures per number of transactions, failures per number of computer runs.

Failure Severity - A rating system for the impact of every recognized credible software failure mode. For example,

- Severity #1 - Loss of life or system
- Severity #2 - Affects ability to complete mission objectives
- Severity #3 - Workaround available, therefore minimal effects on procedures (mission objectives met)
- Severity #4 - Insignificant violation of requirements or standards, not visible to user in operational use
- Severity #5 - Cosmetic issue which should be addressed or tracked for future action, but not necessarily a present problem.

Fault - (1) A defect in the code that can be the cause of one or more failures. (2) An accidental condition that causes a functional unit to fail to perform its required function. Synonymous with bug.

Fault Tolerance - The survival attribute of a system that allows it to deliver the required service after faults have manifested themselves within the system.

Firmware - (1) Computer programs and data loaded in a class of memory that cannot be dynamically modified by the computer during processing. (2) Hardware that contains a computer program and data that cannot be changed in its user environment. The

computer programs and data contained in firmware are classified as software; the circuit containing the computer program and data is classified as hardware. (3) Program instructions stored in a read-only storage. (4) An assembly composed of a hardware unit and a computer program integrated to form a functional entity whose configuration cannot be altered during normal operation. The computer program is stored in the hardware unit as an integrated circuit with a fixed logic configuration that will satisfy a specific application or operational requirement.

Integration - The process of combining software elements, hardware elements or both into an overall system.

Maximum Likelihood Estimation - A form of parameter estimation in which selected parameters maximize the probability that observed data could have occurred.

Module - (1) A program unit that is discrete and identifiable with respect to compiling, combining with other units and loading; for example, input to or output from an assembler, compiler, linkage editor or executive routine. (2) A logically separable part of a program.

Operational - Pertaining to the status given a software product once it has entered the operation and maintenance phase.

Parameter - A variable or arbitrary constant appearing in a mathematical expression, each value of which restricts or determines the specific form of the expression.

Quality - The totality of features and characteristics of a product or service that bears on its ability to satisfy given needs.

Subsystem - A group of assemblies, components or both combined to perform a single function.

Software Quality - (1) The totality of features and characteristics of a software product that bear on its ability to satisfy given needs; for example, to conform to specifications. (2) The degree to which software possesses a desired combination of attributes. (3) The degree to which a customer or user perceives that software meets his or her composite expectations. (4) The composite characteristics of software that determine the degree to which the software in use will meet the expectations of the customer.

Software Reliability - (1) The probability that software will not cause the failure of a system for a specified time under specified conditions. The probability is a function of the inputs to and use of the system, as well as a function of the existence of faults in the software. The inputs to the system determine whether existing faults, if any, are encountered. (2) The ability of a program to perform a required function under stated conditions for a stated period of time.

Software Reliability Engineering - the application of statistical techniques to data collected during system development and operation to specify, predict, estimate, and assess the reliability of software-based systems.

Software Reliability Estimation - The application of statistical techniques to observed failure data collected during system testing and operation to assess the reliability of the software.

Software Reliability Model - A mathematical expression that specifies the general form of the software failure process as a function of factors such as fault introduction, fault removal and the operational environment.

Software Reliability Prediction - A forecast of the reliability of the software based on parameters associated with the software product and its development environment.

System - (1) A collection of people, machines and methods organized to accomplish a set of specific functions. (2) An integrated whole that is composed of diverse, interacting, specialized structures and subfunctions. (3) A group or subsystem united by some interaction or interdependence, performing many duties but functioning as a single unit.

3.0 REFERENCE DOCUMENTS

This section contains reference documents that are applicable to the field of software reliability engineering.

3.1 Primary Documents

The following list of standards should be reviewed prior to implementation of this handbook:

- ANSI / IEEE Std 729-1991, "IEEE Standard Glossary of Software Engineering Terminology"
- MIL-Std 756, "Reliability Modeling and Prediction"

3.2 Other Documents

The following list of documents provide additional information applicable to the scope of the handbook.

- IEEE Std 982.1-1988, "IEEE Standard Dictionary of Measures to Produce Reliable Software"
- IEEE Std 982.2-1988, "IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software"
- IEEE Std 1061-1992, "IEEE Standard for a Software Quality Metrics Methodology"
- MIL-Std-785, "Reliability Programs for Systems and Equipment"
- IEEE Std 1074, "Standard for Life-cycle Processes"
- MIL-HDBK 217, "Reliability Prediction of Electronic Equipment"

4.0 SOFTWARE RELIABILITY MODELING - OVERVIEW, CONCEPTS, AND ADVANTAGES

Software is a complex intellectual product. Inevitably, some errors are made during requirements formulation as well as during designing, coding and testing the product. The development process for high-quality software includes measures that are intended to discover and correct faults resulting from these errors, including reviews, audits, screening by language-dependent tools and several levels of test. Managing these errors involves describing, classifying and modeling the effects of the remaining faults in the delivered product and thereby helping to reduce their number and criticality.

Dealing with faults costs money. It also impacts development schedules and system performance (through increased use of computer resources such as memory, CPU time and peripherals requirements). As is usually recognized, there can be too much as well as too little effort spent dealing with faults. Thus the system engineer (along with management) can use reliability estimation and prediction to understand the current status of the system and make tradeoff decisions.

This section describes the basic concepts involved in software reliability engineering and addresses the advantages and limitations of software reliability prediction and estimation.

4.1 Basic Concepts

There are at least two significant differences between hardware reliability and software reliability. First, software does not fatigue, wear out or burn out. Second, due to the accessibility of software instructions within computer memories, any line of code can contain a fault that, upon execution, is capable of producing a failure.

A software reliability model specifies the general form of the dependence of the failure process on the principal factors that affect it: fault introduction, fault removal and the operational environment.

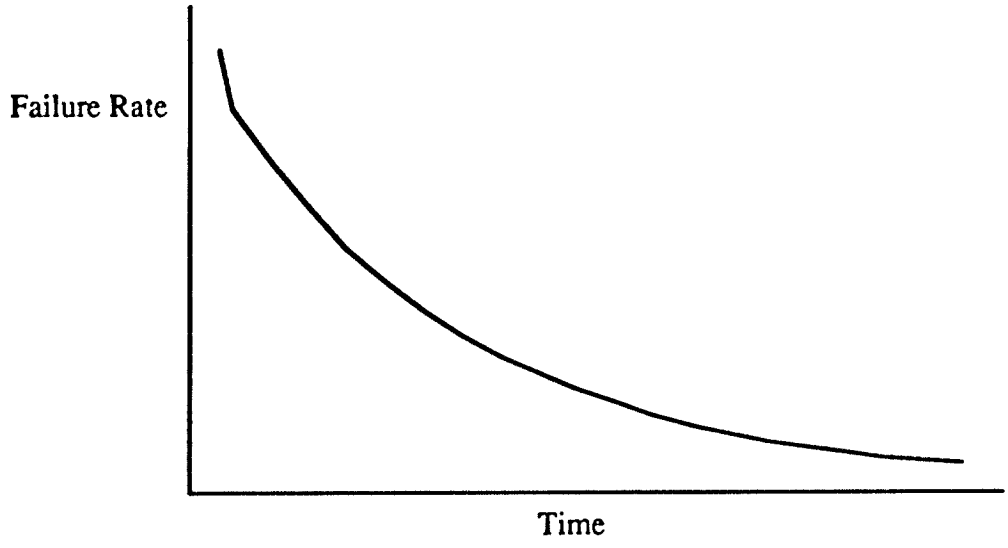

Figure 2 Software Reliability Measurement Curve

The failure rate (failures per unit time) of a software system is generally decreasing due to fault identification and removal. At a particular time, it is possible to observe a history of the failure rate of the software. Software reliability modeling is done to estimate the form of the curve of the failure rate by statistically estimating the parameters associated with the selected model. The purpose of this measure is two-fold: (1) to estimate the extra execution time required to meet a specified reliability objective and (2) to identify the expected reliability of the software when the product is released. This procedure is important for cost estimation, resource planning, schedule validation and quality prediction for software maintenance management.

4.2 Limitations of Software Reliability Prediction and Estimation

There are two types of models that can be applied for software reliability measurement. First, there are prediction models which make use of parameters associated with the software product and its development environment to predict the reliability of a software product. Second, there are estimation models which apply statistical techniques to the observed failures during software testing and operation to forecast the product's reliability. This section describes the limitations of each type of model.

Both prediction and estimation models need good data if they are to yield good forecasts. Good data implies accuracy (that data is truthfully recorded at the time the events occurred) and pertinence (that data relates to an environment that is equivalent to the environment for which the forecast is to be valid). A negative example with respect to accuracy is the restricting of failure report counts to those which are completely filled out. This is negative because they may represent a biased sample of the total reports. A negative example with respect to pertinence would be the use of data from early test runs at an uncontrolled workload to forecast the results of a later test executed under a highly controlled workload.

4.2.1 Prediction Model Advantages / Limitations

In prediction models, the failure probability of a program in development is forecast by comparing it to the known failure probability (or other reliability parameters) of an existing program. The existing program is known as a *proof* program. The advantage of this procedure is that it can be performed at any time during the development, whereas reliability estimation depends on the availability of operational or test data. The validity of the prediction depends on (a) the degree of similarity between the program under development and the proof program

(for which failure rates are known), and (b) the quality of known failure rate data.

When there is direct equivalence between the proof program and the program under development, reliability prediction is a specific application of the Similar Item Method as defined in MIL-STD-756B. The criteria established in MIL-STD-756B for application of this method include:

- Design similarity
- Similarity of service use profile
- Procurement and project similarity
- Proof of reliability achievement.

Because all these criteria can be met only under rare circumstances, alternative methods are usually followed. The most applicable alternative for software involves the following steps:

(1) Estimate the size of the source code. This is a routine step in software development. Many organizations have a size growth model that compensates for the usual underestimating of program size during early stages of development.

(2) Estimate the fault density (faults per line of source code) at the start of formal test (a test activity applicable to the program as a whole and for which computer usage hours will be collected). The preferred approach is to use the fault density determined for a similar program created in the same environment. Where this is not possible, a fault density ranging from 0.001 (for programs developed in a highly disciplined environment and by programmers that have extensive background in the specific application) to 0.01 in a more routine environment may be assumed [MUSA87, Table 5.2].

3) The product of (1) and (2), gives the expected number of faults in the code at the start of formal test. This number corresponds to ω_0 in the Musa Basic Model [MUSA87, Eq. 5.2] and to N in the Jelinski-Moranda Model (Appendix A).

In some environments, the relation between the failure rate at a given point in the development and the fault content at the start of test may be known from experience. In that case, the local factor should be used and the following steps can be omitted.

(4) The key considerations for most models are: (a) the initial number of faults, (b) the probability of executing a specific fault during a single execution (the fault exposure ratio), and (c) the time for which the prediction is to be valid. The latter consideration is at the user's discretion; in some models the time is defined in terms of the number of faults that have been found. The value of the fault exposure ratio is $4.0 \pm 2 \times 10^{-7}$ for 8 out of 13 examples shown in [MUSA87, Table 5.6]; the total range is 1.41×10^{-7} to 10.6×10^{-7}. Where the fault exposure ratio for similar programs is known, that value should be used in preference to the default values of the previous sentence.

(5) The failure probability per fault and unit time is denoted by ϕ in the Jelinski-Moranda Model and by fK in the Musa Basic Model. The factor f is the frequency at which a given (object) instruction will be accessed by the program. It can be computed from f = r/I, where r is the execution rate of the computer and I is the number of object instructions in the program. The dimension of r is instructions per unit time and the time units must be consistent with those for which the failure rate prediction is to be generated. Since execution rate is normally expressed per second and failure rates are expressed per hour, an appropriate conversion has to be performed.

(6) The initial failure rate can then be predicted as $\lambda_0 = fK\omega_0$ for the Musa Basic Model. With these parameters, the expected failure rate at a future point in

time (or after a given number of faults have been detected) can be found by using most of the models described in Section 6 or Appendix A.

Other prediction models use data on the application area, development and test environments, and characteristics of the code (e.g., complexity, modularity) [MCCA87]. These are alternative ways of estimating the fault density and / or the fault exposure ratio. To date, none of these approaches has been shown to be widely applicable. Their use should be restricted to environments where their validity has been demonstrated.

4.2.2 Estimation Model Advantages / Limitations

The premise of most estimation models is that the failure rate is a direct function of the number of faults in the program and that the failure rate will be reduced (reliability will be increased) as faults are detected and eliminated during test or operations. This premise is reasonable for the typical test environment and it has been shown to give credible results when correctly applied. However, the results of estimation models will be adversely affected by:

- Change in failure criteria
- Significant changes in the code under test
- Significant changes in the computing environment.

All of these factors will require a new set of reliability model parameters to be computed. Until these can be established, the effectiveness of the model will be impaired. Estimation of new parameters depends on the measurement of several execution time intervals between failures.

Major changes can occur with respect to several of the above factors when software becomes operational. In the operational environment, the failure rate is a function of the fault content of the program, of the variability of input and computer states, and of software maintenance policies. The latter two factors are under management control and are frequently utilized to achieve an expected or desired range of values for the failure rate or the downtime due to software causes. Examples of management action that decrease the failure rate include: avoidance of high work loads and avoidance of data combinations that have caused previous failures [GIFF84,IYER83]. Software in the operational environment may not exhibit the reduction in failure rate with execution time that is an implicit assumption in most estimation models [HECH86a]. Knowledge of the management policies is therefore essential for selection of a software reliability estimation procedure for the operational environment. Thus, the estimation of operational reliability from data obtained during test may not hold true during operations.

Another limitation of software reliability estimation models is their use for verifying ultra-high requirements. For example, if a program executes successfully for x hours, there is maybe a 0.5 probability that it will survive the next x hours without failing [LITT90]. Thus, to have the kind of confidence needed to verify a 10^{-9} requirement would require that the software execute failure-free for several billion hours. Clearly, even if the software had achieved such a reliability, one could never assure that the requirement was met. The most reasonable verifiable requirement is somewhere in the 10^{-3} or 10^{-4} range.

It is important to understand the nature of the program when discussing ultra-high requirements. Many ultra-reliable applications are implemented on relatively small, slow, inexpensive computers. Furthermore, the critical programs are small (less than 1000 source lines of code) and execute infrequently during an actual mission. With this knowledge, it may be feasible to test the critical program segment on several faster machines, considerably reducing the required test time.

Furthermore, where very high reliability requirements are stated (failure probabilities $< 10^{-6}$) they frequently are applicable to a software controlled process together with its protective and mitigating facilities and therefore they tend to be overstated if applicable to

the process alone. An example of a protective facility is an automatic cut-off system for the primary process and reversion to analog or manual control. An example of a mitigation facility is an automatic sprinkler system that significantly reduces the probability of fire damage in case the software controlled process generates excessive heat. If the basic requirement is that the probability of extensive fire damage shall not exceed 10^{-6} per day, and if both protecting and mitigating facilities are in place, it is quite likely that further analysis will show the maximum allowable failure rate for the software controlled process to be on the order of 10^{-3} per day and hence within the range of current reliability estimation methods.

Where the requirements for the software controlled process proper still exceed the capabilities of the estimation methodology after allowing for protective and mitigating facilities, fault tolerance techniques may be applied. These may involve fault tolerance [HECH86b] or functional diversity. An example of the latter is to control both temperature and pressure of steam generation, such that neither one of them can exceed safety criteria. The reduction in failure probability that can be achieved by software fault tolerance depends in a large measure on the independence of failure mechanisms for the diverse implementations. It is generally easier to demonstrate the independence of two diverse functions than it is to demonstrate the independence of two computer programs, and hence functional diversity is frequently preferred.

5.0 SOFTWARE RELIABILITY ESTIMATION PROCEDURE

This section provides guidance to the practitioner on how to do software reliability estimation and what types of analysis can be performed using the technique. It defines a generic step-by-step procedure for executing software reliability estimation and describes possible analysis using the results of the estimation procedure.

5.1 Generic Procedure

An eleven-step generic procedure for estimating software reliability is listed below. This generic procedure should be tailored to the project and the current life-cycle phase. Some steps will not be used in some applications, but the structure provides a convenient and easily remembered standard approach. The following steps can be used to generate a checklist for reliability programs:

1) Identify Application

2) Specify the Requirement

3) Allocate the Requirement

Since this document is concerned only with the test through operational life-cycle activities, only steps (4) through (11) are discussed.

4) Define Failure

5) Characterize the Operational Environment

6) Select Tests

7) Select Modes

8) Collect Data

9) Estimate Parameters

10) Validate the Model

11) Perform Analysis

5.1.1 Define Failure

A *project specific* failure definition is usually negotiated by the testers, developers, and users. It is agreed upon prior to the beginning of test. In spite of this necessary tailoring, there are often commonalities in the definition among similar products (e.g., most people agree that a software bug that when encountered stops all processing is a failure). The important consideration is that the definition be consistent over the life of the project.

There are a number of specific considerations

relating to the interpretation of failure for systems. The analyst must determine the answers to these questions:

- Is a failure counted if it is consciously decided not to seek out and remove the cause of a particular failure?
- Are repeated failures counted?
- What is a failure in a fault-tolerant system?
- Are a series of failures counted if they are triggered by data degradation?

A discussion of each of these considerations is provided in [MUSA87, pp 77-85].

Projects need to classify failures by their severity. An example classification is provided in Section 2. Classes are usually separated by an order of magnitude in costs. Impact can not ordinarily be estimated with great precision. It is desirable to consider severity by type, and by user requirement.

For some projects, there appears to be a relative homogeneity with respect to time-of-failure among high-severity failures. For example, if 10 percent of the failures occurring early in test fall in a particular class, about the same percentage will be expected to be found in that class late in test. This permits making, for example, statistical estimates based on all data to achieve a smaller confidence interval and then adjusting them to *per class* estimates. It also is possible to weight failure data by a variable (such as cost) associated with class and to obtain compound estimates such as failure cost rather than failure intensities.

It is recommended that failure times be recorded in execution time. However, should execution time not be readily available, elapsed clock time is a satisfactory approximation if machine utilization is constant (when averaged over a time period comparable to the times between failures). If utilization is not constant, one often can weight the clock time by a measure that is proportional to the utilization, such as number of uses of a real-time system. Execution time also can be approximated by natural units like transactions.

When failure times are collected from multiple machines functioning simultaneously, intervals between failures should be counted by considering execution time on all machines. If the machines have different average instruction execution rates, execution times should be adjusted to a reference machine [MUSA87, pp 162-165].

5.1.2 Characterize the Operational Environment

Characterization of the operational environment has three aspects: 1) system configuration, 2) system evolution, and 3) system operational profile.

System configuration is the arrangement of the system's components. Software-based systems are just that; they can not be pure but must include hardware as well as software components.

Distributed systems are a type of system configuration. The purpose of determining the system configuration is twofold:

- To determine how to allocate system reliability to component reliabilities
- To determine how to combine component reliabilities to establish system reliability [MUSA87, pp 85-106].

In modeling software reliability, it is necessary to recognize that systems frequently evolve as they are tested. That is, new code or even new components are added. Special techniques for dealing with evolution are provided in [MUSA87, pp 166-176].

The system operational profile characterizes in quantitative fashion how the software will be used. It lists all operations realized by the software and the probability of occurrence and criticality of each operation.

A system may have multiple operational profiles or operating modes. They usually represent difference in function associated with significant environmental variables. For example, a space vehicle may have ascent, on-

orbit and descent operating modes. Operating modes may be related to time, installation location, customer or market segment. Reliability can be tracked separately for different modes if they are significant. The only limitation is the extra data collection and cost involved.

5.1.3 Select Tests

Many applications of software reliability engineering involve the execution of operations and collection of failure data. Operations should be picked to reflect how the system will actually be used. Reference Appendix C for information that may be useful in determining failure rates. In other words, the test operational profile should represent the field operational profile.

The tester selects one of the following approaches:

- Test duplicates actual operational environments (as closely as possible)
- Testing conducted under more severe conditions; for extended periods of time - resulting in failures being accumulated in less than expected time.

The modeling effort must take into account the specific approach taken by the test team to expose faults so that accurate forecasts can be made.

5.1.4 Select Models

The models described in Section 6 have been identified for giving good results in specific environments, but it can not be guaranteed that they will be suitable in new environments. Therefore it is recommended that each user compare several models prior to final selection.

A list of the model selection criteria described in Section 6.1 is provided below:

- Predictive Validity
- Ease of Parameter Measurement
- Quality of Assumptions
- Capability
- Applicability
- Simplicity
- Insensitivity to Noise

In general, each model should be evaluated by these criteria, using the best model to make forecasts.

5.1.5 Collect Data

Data collection must be geared toward the overall objectives of the software reliability effort, such as the attainment of a failure-free interval.

In considering setting up a reliability program, one must avoid several pitfalls. The first is that every bit of information about the program and what happens to it as it evolves over the life cycle needs to be kept. The second is that clearly defined objectives for the data collection process are not necessary. These two pitfalls result in too much effort expended with too little payback. When a massive amount of data is required, it is usually the program manager's people that are impacted. Cost and schedule suffer.

Two additional points that should be kept in mind while planning to collect data and collecting data are: (1) motivate the data collectors, and (2) review the collected data promptly. If these two things are not done, quality will suffer.

A list of the data collection steps detailed in Section 7.1 is provided below:

1) Establish the objectives.
2) Set up a plan for the data collection process.
3) Apply tools.
4) Provide training.
5) Perform trial run.
6) Implement the plan.

7) Monitor data collection.

8) Evaluate the data as the process continues.

9) Provide feedback to all parties.

In general, a process should be established addressing each of these steps, and a successful software reliability data collection program will emerge.

5.1.6 Estimate Parameters

There are three common methods of parameter estimation: method of moments, least squares, and maximum likelihood. Each of these methods has attributes that make it useful. However, maximum likelihood estimation is the most commonly used approach. A full treatment of parameter estimation is provided in [MUSA87, FARR83, and SHOO83]. All of the software reliability engineering tools described in Appendix B perform parameter estimation as one of their capabilities using one or more of these methods.

5.1.7 Validate the Model

Several considerations are involved in properly validating a model for use on a given production project. First, it is necessary to deal with the assumptions of the model under evaluation. Choosing appropriate failure data items and relating specific failures to particular intervals of the life-cycle or change increments often facilitate this task [SCHN92]. Depending on the progress of the *production* project, the model validation data source should be selected from the following, listed in the order of preference:

1) *Production* project failure history (if project has progressed sufficiently to produce failures).

2) Prototype project employing similar products and processes as the *production* project.

3) Prior project employing similar products and processes as the *production* project (reference Appendix C)

Using one of these data sources, the analyst should execute the model for several specific times within the failure history period and then compare the model output to the actual subsequent failure history using one of the following:

1) Predictive validity criteria (Section 6.1.1).

2) A traditional statistical goodness-of-fit test (e.g., Chi-square or Kolmogorov-Smirnov).

It is important that a model be continuously re-checked for validation, even after selection and application, to ensure that the fit to the observed failure history is satisfactory. In the event that a degraded model fit is experienced, alternate candidate models should be evaluated using the procedure above.

5.1.8 Perform Analysis

Once the data has been collected and the model parameters estimated, the analyst is ready to perform the appropriate analysis. This analysis may be to estimate the current reliability of the software, forecast the number of faults remaining in the code, or forecasting a testing completion date. Section 5.2 details a set of common analyses conducted using software reliability theory.

One pitfall to be careful of is the combination of a software reliability value into a system reliability calculation. If the analysis calls for producing a system reliability figure and the software reliability is calculated in terms of execution time, it must be converted to calendar time for combination with hardware reliabilities to calculate the system value.

5.2 Recommended Analysis Practice

This section provides details of analysis procedures for some common engineering or management activities that can be aided by software reliability engineering technology. These details are in most cases a description of the analysis that must be performed as the last step of the generic procedure described in Section 5.1. Although this list is far from complete, it is a set to start from.

5.2.1 Estimate Current Reliability

Since software will not fail until the software is executed and a software fault is manifested by the computer, the time measurement based on CPU time for failure data collection is preferred. However, there are approximating techniques if the direct measurement of CPU time is not available [MUSA87, pp 156-158]. When combined with hardware reliability measurement (to form the system reliability prediction) the CPU time also can be transformed to calendar time [MUSA87, pp 113-139].

Reliability estimations in test and operational phases basically follow the same procedure. However, there is a difference. During the testing phase, software faults are intended to be removed as soon as the corresponding software failures are detected. As a result, the reliability growth could be observed. However, in the operational phase, correcting a software fault involves changes of multiple software copies in the customers' sites, which, unless the failure is catastrophic, is not always done until the next software release.

Therefore, the software failure rate usually remains constant until a new version is released, in which case a jump in reliability should be observed. Nevertheless, the users might change the use of the software to avoid triggering the known failure. In other words, the operational profile is changed and certain growth of reliability could still be observed.

5.2.2 Forecast Achievement of a Reliability Goal

The date at which a given reliability goal can be achieved is obtainable from the software reliability modeling process illustrated in Figure 3. As achievement of the reliability target approaches, the adherence of the actual data to the model should be reviewed and the model calibrated if necessary. Refer to Appendix F, "Using Reliability Models for Developing Test Strategies."

5.2.3 Forecast Additional Test Duration

Additional test duration may be predicted if the initial and objective failure intensities and the parameters of the model are known. (These are identified for each model in Section 6.) For the Musa Basic exponential model we have:

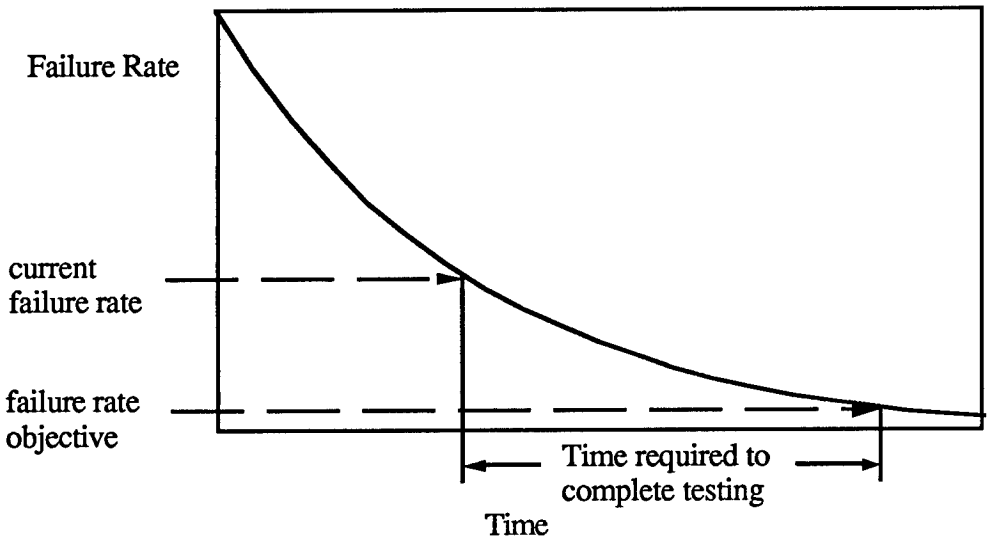

Figure 3 Example Software Reliability Measurement Application

$$\Delta t = \frac{\nu_0}{\lambda_0} \ln\left(\frac{\lambda_0}{\lambda_F}\right)$$

where Δt is the test duration in CPU hr, ν_0 is the total failures parameter of the model, λ_0 is the initial failure intensity, and λ_F is the objective failure intensity.

The formula for the Musa-Okumoto Logarithmic Poisson model is

$$\Delta t = \frac{1}{\theta}\left(\frac{1}{\lambda_F} - \frac{1}{\lambda_0}\right)$$

where θ is the failure intensity decay parameter.

Calendar time test duration could be computed manually. However, all calculations are generally available in software reliability tools (See Appendix B), and the formulas given above are only occasionally applied manually.

5.2.4 Establish Conformance with Acceptance Criteria

If reliability-related criteria are part of software acceptance, the model should be selected so that its results can be easily interpreted for conformance with the selection criteria. For example, time-to-failure models are consistent with requirements for failure-free intervals. Failure count models are suitable for establishing conformance with maximum failure rate requirements.

5.2.5 Manage Introduction of New Features into Operational Software

Decisions about whether and when to introduce new features into operational software must be made. Introduction of new features carries the risk of adding new faults and hence increasing the failure intensity. This could raise the failure intensity to such a level that the impact on service is unacceptable. Software reliability engineering provides a quantitative way of measuring service and hence a guide for permitting or delaying the introduction of the new features. Failures are regularly recorded during operation and failure data is entered in a software reliability estimation program, which is run at regular intervals (frequently weekly). A running plot of failure intensity is generated.

A failure intensity objective is selected based on a balance between the need for new features and the need that old features operate reasonably reliably. The proximity of the actual failure intensity to the objective is now used as the criterion for accepting new features. New features are accepted only when the actual failure intensity is sufficiently below the objective that it appears unlikely that the addition of the new features will increase the failure intensity substantially above the objective.

5.2.6 Evaluate Reliability Impact of Software Engineering Technology Variables

It is important to know the impact of technology on software reliability. This knowledge will make it possible to design an efficient development process for a particular software product. These impact studies have not been performed to any extent at the writing of this document, but they could and should be. For example, the relationship between effort devoted to design inspection per thousand source lines of code and the change in failure intensity should be studied. This is done by holding other variables constant as design inspection effort is varied. The resultant quantity that is measured could be initial failure intensity at the start of system test.

5.2.7 Estimate Maintenance Staffing Requirements

Three quantities are needed to estimate the staff required to restore systems after a software failure: first, the average time required for a repair person to restore the system after a failure (including travel time); second, the expected operating time of the software in time units; and third, the expected failure rate of the software in operation.

Multiplying the failure rate by the operational

time yields the expected number of failures per time unit. Using this number and the average time to restore the system, the number of repair personnel can be derived. It is important to assign more repair personnel than this estimate to account for variations in failure occurrence that may result in lower system availability.

5.2.8 Assist Safety Certification

A software safety failure can be defined as any software system behavior that involves risk to human life, risk of injury or risk of equipment damage. Thus, Failure Severity #1 (see Terminology Section 2 - "Failure Severity") failures fall into this category. The failure rate based on the failures in this category can be determined to support software safety certification. It is important to note that reliability is a necessary, although not sufficient, condition to ensure safety and should not be used as the only criterion for safety certification.

6.0 SOFTWARE RELIABILITY ESTIMATION MODELS

There are many ways to develop a software reliability model: (a) describe it as a stochastic process, (b) relate it to a Markov model, (c) define the probability density or distribution function, or (d) specify the hazard function. These approaches are all equivalent and equally correct. There are three general classes of software reliability estimation models: Exponential non-homogeneous Poisson process (NHPP) models, Non-exponential NHPP models and Bayesian models. The following paragraphs describe the characteristics of each general class.

Exponential NHPP Models

Exponential NHPP models use the stochastic process and the hazard function approach. The hazard function, z(t), is generally a function of the operational time, t. Several different derivations of z(t) are given in [SHOO90a]. The probability of success as a function of time is the reliability function, R(t), which is given by:

$$R(t) = \exp\left[-\int_0^t z(y)\,dy\right]$$

Sometimes reliability is expressed in terms of a single parameter: mean time to failure, (MTTF). MTTF is given by:

$$MTTF = \int_0^\infty R(t)\,dt$$

On occasion the reliability function may be of such a form that MTTF is not defined. The hazard function (or failure intensity, [MUSA87, pp. 11, 18]) or the reliability function can be used in this case. The hazard function can be constant or can change with time.

Representative models selected for this class include: Shooman's model; Musa's Basic model; Jelinski and Moranda's model (described in Appendix A); and the generalized exponential model (described in Section 6.2). Model objectives, assumptions, parameter estimates, and considerations for using the model are described in the appropriate section.

Non-Exponential NHPP Models

Non-Exponential NHPP models also use the stochastic process and the hazard function approach. They are generally applicable when testing is done, according to an operational profile that is not uniform in nature. Early fault corrections have a larger impact on the failure intensity function than later ones.

Representative models selected for this class include: Duane's model; Brooks and Motley's Binomial and Poisson models; Yamada's S-shaped model (all described in Appendix A); and Musa and Okumoto's Logarithmic Poisson (described in section 6.2). The assumptions and format of the respective model, its estimates for model fitting, and finally considerations for the employment of the model are described in the appropriate section.

Bayesian Models

Bayesian models differ from NHPP models in two ways. First, where NHPP models only allow for change in reliability when a fault is discovered and corrected, Bayesian models allow reliability to change based on the length of failure-free testing time periods. Second, NHPP models assume that the hazard function is directly proportional to the number of faults in the program and hence the reliability is a function of this fault count. The Bayesian approach argues that a program can have many faults in unused sections of the code and exhibit a higher reliability than software with only one bug in a frequently exercised section of code. Representative models of this class are those developed by Littlewood [LITT79].

6.1 Criteria for Model Evaluation

This following criteria should be used for conducting an evaluation of software reliability models in support of a given project.

- Model predictive validity: the performance and correctness of the forecast quality of each model. Measures defined for this are: accuracy, trend, bias, and noise.
- Ease of measuring parameters: the resource requirement and impact of measuring parameters for each model, including cost, schedule impact for data collection, and physical significance of parameters to the software development process.
- Quality of assumptions: the closeness to the real world, and adaptability to a special environment.
- Capability: the ability of each model to estimate useful quantities needed by software project personnel, including expected MTTF, time to reach a specified MTTF goal, and the required resources to reach that goal.
- Applicability: the ability to handle program evolution and change in test and operational environment.
- Simplicity: ease of understanding the concept, data collection, program implementation, and validation.
- Insensitivity to noise: the ability of the model to produce results in spite of small differences in input data and parameters without losing responsiveness to significant differences

6.1.1 Model Predictive Validity

To compare a set of models on a given set of failure data, one must examine which of the *fitted* models is best in agreement with the observed data. A *fitted* model is one that has had its parameters estimated from the observed data. The question being asked is: Is it plausible to have obtained the observed data by sampling from the fitted model? If $\hat{F}$ is the function of the model with estimated parameters, this question can be answered by a hypothesis test with a null hypothesis:

H_0 : the failure data are from a model with distribution function, $\hat{F}$.

This is called a goodness-of-fit test since it tests how well the model "fits" the observed data. Goodness-of-fit tests are a way to detect systematically fairly gross disagreement between the data and the fitted model. The literature on goodness-of-fit tests is quite extensive; the chi-square and Kolmogorov-Smirnov tests are the most popular tests [HOEL71].

In addition to these techniques for assessing model fit, the following four measures can be used to compare model forecasts on a set of failure data:

6.1.1.1 Accuracy

Forecasting accuracy is measured by the prequential likelihood (PL) function [LITT86]. Let the observed failure data be a sequence of times $t_1, t_2,..., t_{i-1}$ between successive failures. The objective is to use the data to forecast the future unobserved T_i. More precisely, we want a good estimate of $F_i(t)$, defined as $P(T_i,<t)$, i.e., the probability that T_i is less than a variable t. The

forecasting distribution $\tilde{F}_i(t)$ for T_i based on $t_1, t_2, ..., t_{i-1}$ will be assumed to have a pdf (probability density function).

$$f_i(t) = \frac{d}{dt}\tilde{F}_i(t)$$

For such one-step-ahead forecasts of $T_{j+1},...T_{j+n}$, the prequential likelihood is:

$$PL_n = \prod_{i=j+1}^{j+n} \tilde{f}_i(t_i)$$

Since this measure is usually very close to zero, its natural logarithm is frequently used for comparisons. Given two competing software reliability models A and B, the prequential likelihood ratio is given by

$$PLR_n = \frac{P\ln(A)}{P\ln(B)}$$

The ratio represents the likelihood that one model will give more accurate forecasts than the other model. If $PLR_n \rightarrow \infty$ as $n \rightarrow \infty$, model A is favored over model B.

6.1.1.2 Bias

A model is considered *biased* if it forecasts values that are consistently longer than the observed failure times, or consistently shorter than the observed times. To measure the amount of a model's bias, one can compute the maximum vertical distance (i.e., the Kolmogorov Distance [HOEL71]) between the line of unit slope and the values of the probability integral transformation given by:

$$u_i = \tilde{F}_i(t_i)$$

Each u_i is a probability integral transform of the observed t_i using the previously calculated predictor $\tilde{F}_i$ based upon $t_1, t_2, ..., t_{i-1}$. That is, u_i is the estimated model distribution function evaluated at the observed failure times. To identify the direction toward which a model is biased, use the notation that a positive number means that the model tends to be optimistic, while a negative number represents the model to be pessimistic. In either case, the smaller the absolute value of the number is, the less bias there is inherent in the model.

6.1.1.3 Trend

In some cases a model may be optimistic in an early set of forecasts and pessimistic in a later set of forecasts. The bias test described above will average these effects, and the model will appear unbiased. In this case, it is important to examine the u_i's for trend. Trend is defined as the Kolmogorov Distance between the line of unit slope and the cdf of y_i, where

$$x_i = -\ln(1-u_i)$$

$$y_i = \frac{\sum_{j=1}^{i} x_j}{\sum_{j=1}^{n} x_j} \quad \text{for } i \le n$$

The trend represents the consistency of the model's bias. A small value means that the model is more adaptable to changes in the data behavior, and hence yields better performance.

6.1.1.4 Noise

The test for noise is roughly analogous to the mean square error in classical statistics. The goal of the measure is to indicate objectively which model is giving the least variable forecasts (i.e., finding the most stable model for a particular data set). The measure is defined as

$$\text{Noise} = \sum_{i=2}^{n} \left| \frac{r_i - r_{i-1}}{r_{i-1}} \right|$$

where r_i is the forecasted failure rate ($1/T_i$). Note that the forecasted median of the failure time distribution, denoted by m_i, may be used in place of r_i. In either case, small values represent less noise in the forecasts of the model, indicating better smoothness. A

Noise value equal to infinity indicates that a failure rate of zero has been forecasted by the model.

6.1.2 Ease of Measuring Parameters

Ease of measuring parameters refers to the number of parameters a model requires, and the difficulties in estimating these parameters. Most software reliability estimation models incorporate either two or three parameters. As a rule-of-thumb, a model requires failure data equal to at least five times the number of parameteres to be estimated. In general, a three-parameter model can achieve better accuracy in fitting the failure data curve than can a two-parameter model. However, this is not generally true for making software reliability forecasts. When two models demonstrate the same level of forecasting capability, the model which requires fewer parameters should be chosen. This is not only because a model with fewer modes is easier to apply, but also because a software reliability engineer can more successfully interpret the physical significance of the parameters to provide appropriate feedback to the software development process.

6.1.3 Quality of Assumptions

The assumptions that a software reliability model makes should be as close to the real project testing and operational situation as possible. Common assumptions made in the software reliability models are:

- Test input randomly encounter faults.
- The effects of all failures are independent.
- The test space "covers" the use space.
- All failures are observed when they occur.
- Faults are immediately removed upon failure or not counted again.
- The software failure rate is related to the number of software faults remaining in the software; software reliability models specify this relationship.

If an assumption is testable, it should be supported by data to validate the assumption. If an assumption is not testable, it should be examined through the viewpoint of logical consistency and software engineering experience. Moreover, all model assumptions should be judged by their clarity and explicitness. This will help to determine whether a particular model applies to the current project.

6.1.4 Capability

Capability refers to the ability of a model to estimate reliability related quantities for software systems. These quantities include:

- The present reliability of the software, the software failure rate, or mean-time-to-failure (MTTF), or the failure rate distribution.
- Confidence intervals for all estimated parameters.
- Expected date of achieving a specified reliability, failure rate, or MTTF objective.
- Resource (human and computer) and cost estimates related to achieving the reliability objective.

Other than the capability to make software reliability measurements in the testing and operational phase, the capability of a model to make software reliability predictions in the system design and early development phases is also very important. These predictions should be examined through future research in software metrics, the software development environment, and the operational profile.

6.1.5 Applicability

Applicability of the software models should be examined through various sizes, structures, functions, and application domains. An advantage of a specific model is its usability in different development and operational environments, and different life-cycle phases. In the application of software reliability models, the following situations should be dealt with by the models:

- Evolving software (i.e., software that is incrementelly integrated during testing),

- Classification of failure severity,
- Incomplete failure data,
- Hardware execution rate differences,
- Multiple installations of the same software,
- Project environments departing from model assumptions.

6.1.6 Simplicity

Simplicity refers to three aspects of a model: the data collection process, the modelling concept, and its implementation by a software tool. Simplicity in data collection reduces the measurement cost, increases the data accuracy, and makes it easier for model application. Simplicity in the modeling concepts makes it easier to understand the assumptions, estimate the parameters, apply the models, and interpret the results. Simplicity in the model implementation encourages an efficient use of computers to facilitate the model applications which are normally computationally intensive.

In choosing a model, one should give weight to simplicity. Until an organization has practiced reliability estimation a few times, no more complex models are warranted, nor in general will there be data to support more complex models.

6.1.7 Insensitivity to Noise

Software reliability data generally contain noise irrelevant to the modeling process. The most common source of noise is that software failure data is recorded in project calendar time rather than in software execution time. Even when software failures are tracked carefully based on execution time, the software testing process may be inconsistent with the model assumptions (e.g., the software is not tested randomly). Therefore, a model should demonstrate its validity in an ideal situation as well as in situations when the failure data is incomplete or contains measurement uncertainties.

6.2 Recommended Models

The following models are recommended as initial models for software reliability estimation; the order is arbitrary: the Schneidewind model, the generalized Exponential model, the Musa / Okumoto Logarithmic Poisson model and the Littlewood / Verrall model. If these models can not be validated (see Section 5.1.7) or do not meet the criteria defined in Section 6.1 for the project, alternative models are described in Appendix A.

6.2.1 Recommended Model: Schneidewind Model

6.2.1.1 Schneidewind Objectives

The objectives of this model are to forecast the following software product attributes:

- Number of failures that will occur by a given time (execution time, labor time, or calendar time)
- Maximum number of failures that will occur over the life of the software
- Maximum number of failures that will occur after a given time
- Time required for a given number of failures to occur
- Number of faults corrected by a given time
- Time required to correct a given number of faults
- Number of outstanding (observed but not corrected) faults at a given time
- Incremental time required to correct a given number of outstanding faults
- Time required for outstanding faults to reach a given value

The basic philosophy of this model is that as testing proceeds with time, the failure detection process changes. Furthermore, recent failure counts are usually of more use than earlier counts in forecasting the future. Three approaches are employed in utilizing

the failure count data, i.e. number of failures detected per unit of time. Suppose there are m intervals of testing and f_i failures were detected in the i^{th} interval, one of the following can be done:

- Utilize all of the failures for the m intervals
- Ignore the failure counts completely from the first s – 1 time intervals ($2 \leq s \leq m$) and only use the data from intervals s through m.
- Use the cumulative failure count from intervals 1 through s – 1, i.e.

$$F_{s-1} = \sum_{i=1}^{s-1} f_i$$

The first approach is applicable when one feels that the failure counts from all of the intervals are useful in predicting future counts. The second approach is to be used when it is felt that a significant change in the failure detection process has occurred and thus only the last $m - s + 1$ intervals are useful in future failure forecasts. The last approach is an intermediate one between the other two. Here it is felt that the combined failure counts from the first $s - 1$ intervals and the individual counts from the remaining are representative of the failure and detection behavior for future forecasts.

6.2.1.2 Schneidewind Assumptions

The assumptions to the Schneidewind model are:

- The number of failures detected in one interval is independent of the failure count in another.
- Only new failures are counted.
- The fault correction rate is proportional to the number of faults to be corrected.
- The software is operated in a similar manner as the anticipated operational usage.
- The mean number of detected failures decreases from one interval to the next.
- The intervals are all the same length.
- The rate of failure detection is proportional to the number of faults within the program at the time of test. The failure detection process is assumed to be a nonhomogeneous Poisson process with an exponentially decreasing failure detection rate. The rate is taken to be of the form

$$d_i = \alpha \exp(-\beta i)$$

for the i^{th} interval where $a > 0$ and $b > 0$ are the constants of the model.

6.2.1.3 Schneidewind Structure

Two parameters are used in the model: α, which is the failure rate at time m=0, and β, which is a proportionality constant that affects the failure rate over time (i.e., small β implies a large failure rate; large β implies a small failure rate). In these estimates: m is the last observed count interval; s is an index of time intervals; X_k is the number of observed failures in interval k; X_{s-1} is the number of failures observed from 1 through s-1 intervals; $X_{s,m}$ is the number of observed failures from interval s through m; and $X_m = X_{s-1} + X_{s,t}$. The likelihood function is then developed as

$$\begin{aligned} \log L = {} & X_m\left[\log X_m - 1 - \log\left(1 - \exp(-\beta m)\right)\right] \\ & + X_{s-1}\left[\log\left(1 - \exp(-\beta(s-1))\right)\right] \\ & + X_{s,m}\left[\log\left(1 - \exp(-\beta)\right)\right] \\ & - \beta \sum_{k=0}^{m-s} (s + k - 1) X_{s+k} \end{aligned}$$

This function is used to derive the equations for estimating α and β for each of the three approaches described earlier. In the equations that follow, α and β are estimates of the population parameters.

Parameter Estimation: Approach 1

Use all of the failure counts from interval 1 through m (i.e., s=1). The following two equations are used to estimate β and α, respectively.

$$\frac{1}{\exp(\beta)-1}-\frac{m}{\exp(\beta m)-1}=\sum_{k=0}^{m-1}k\frac{X_{k+1}}{X_m}$$

$$\alpha=\frac{\beta X_m}{1-\exp(-\beta m)}$$

Parameter Estimation: Approach 2

Use failure counts only in intervals s through m (i.e., $1 \le s \le m$). The following two equations are used to estimate β and α, respectively. (Note that approach 2 is equivalent to approach 1 for s = 1.)

$$\frac{1}{\exp(\beta)-1}-\frac{m-s+1}{\exp(\beta(m-s+1))-1}=\sum_{k=0}^{m-s}k\frac{X_{k+s}}{X_{s,m}}$$

$$\alpha=\frac{\beta X_{s,m}}{1-\exp(-\beta(m-s+1))}$$

Parameter Estimation: Approach 3

Use cumulative failure counts in intervals 1 through s-1 and individual failure counts in intervals s through m (i.e., $2 \le s \le m$). This approach is intermediate to approach 1 which uses all of the data and approach 2 which discards "old" data. The following two equations are used to estimate β and α, respectively. (Note that approach 3 is equivalent to approach 1 for s = 2.)

$$\frac{(s-1)X_{s-1}}{\exp(\beta(s-1))-1}+\frac{X_{s,m}}{\exp(\beta)-1}-\frac{mX_m}{\exp(\beta m)-1}=\sum_{k=0}^{t-s}(s+k-1)X_{s+k}$$

$$\alpha=\frac{\beta X_m}{1-\exp(-\beta m)}$$

Mean Square Error Criterion

The Mean Square Error (MSE) criterion can be used to select one of the three approaches by finding the optimal value of s. The MSE computes the sum of the squared differences between model predictions and actual cumulative failure counts x(i) in the range $s \le i \le m$. The following equation applies to approach 2 above. For approach 1 and approach 3, s=1.

$$MSE=\frac{\sum_{i=s}^{m}\left[\alpha/\beta\left(1-\exp(-\beta(i-s+1))\right)-x(i)\right]^2}{m-s+1}$$

Thus, for each value of s, compute MSE using the above formula. Choose s equal to the value for which MSE is smallest. The result is an optimal triple (β, α, s) for your data set. Then apply the appropriate approach to your data.

6.2.1.4 Schneidewind Limitations

The limitations of the model are the following:

- It does not account for the possibility that failures in different intervals may be related.
- It does not account for repetition of failures.
- It uses intervals of equal length.
- It does not account for the possibility that failures can increase over time as the result of software modifications.

These limitations can be ameliorated by configuring the software into versions that represent the previous version plus modifications. Each version represents a different module for reliability prediction purposes: the model is used to predict reliability for each module.

6.2.1.5 Schneidewind Data Requirements

The only data requirements are the number of errors, f_i, i = 1, ..., m, per testing period.

Although a data base is not required, it would be very useful to create and maintain a reliability data base for several reasons: input data sets could be rerun, if necessary; reliability predictions and assessments could be made for various projects; predicted reliability could be compared with actual reliability for these projects. This data base would allow the model user to perform several useful analyses: to see how well the model is performing; to compare reliability across projects to see whether there are development factors that contribute to reliability; and to see whether reliability is improving over time for a given project or across projects.

6.2.1.6 Schneidewind Applications

The major model applications are described below. These are separate but related uses of the model that, in total, comprise an integrated reliability program.

- Forecasting: Forecasting future failures, fault corrections and related quantities described in section 6.2.1.7.

- Control: Comparing forecast results with pre-defined goals and flagging software that fails to meet those goals.

- Assessment: Determining what action to take for software that fails to meet goals (e.g., intensify inspection, intensify testing, redesign software, revise process). The formulation of test strategies is also part of assessment. Test strategy formulation involves the determination of: priority, duration and completion date of testing, allocation of personnel, and allocation of computer resources to testing.

6.2.1.7 Reliability Forecasts

Using the optimal triple (α,β,s) which were given in section 6.2.1.3, various reliability forecasts can be computed. The approach 2 equations are given where $T \geq s$. For approach 1 and approach 3, s=1 and $T \geq 1$, where T is preferably execution time but can be labor time or calendar time.

- Time to detect a total of F failures, when the current time is t and X(t) failures have been observed

$$T_f(t) = \log\left[\alpha / \left(\alpha - \beta(F(t) - X(t))\right) / \beta\right] - (t - s + 1)$$
$$\text{for } \alpha > \beta(F(t) + X(t))$$

- Forecasted Number of Failures after time T

$$F(T) = (\alpha / \beta)\left[1 - \exp(-\beta(T - s + 1))\right]$$

- Maximum Number of Failures ($T = \infty$)

$$F(\infty) = \alpha / \beta$$

- Maximum Number of Remaining Failures, forecasted at time t, after X(t) failures have been observed

$$RF(t) = \alpha / \beta - X(t)$$

- Faults Corrected after time T

$$C(T) = (\alpha / \beta)\left[1 - \exp\left(-\beta((T - s + 1) - \Delta t)\right)\right]$$

  where Δt is the mean lag in correcting faults after failures have been observed. (Δt can be estimated from the data.)

- Time to correct C faults

$$T_C = \Delta t + \left[\left(\log\left[\alpha / (\alpha - \beta C)\right]\right) / \beta\right] + s - 1$$
$$\text{for } \alpha > \beta C$$

- Outstanding Faults Remaining at time T

$$N(T) = F(T) - C(T)$$

- Outstanding Fault Correction Time

$$\Delta T_N = \left[\log\left(\left(N\beta \exp(\beta(T - s + 1)) / \alpha\right) + 1\right)\right] / \beta$$

where N is the number of faults to correct starting at time T.

- Outstanding Fault Time

  The predicted time for the number of outstanding faults to reach the value N is

$$T_N = \left[\left(\log\left[\left(\alpha \exp(\beta \Delta T_N) - 1\right) / \beta N\right]\right) / \beta\right] + s - 1$$

6.2.1.8 Schneidewind Implementation Status and Reference Applications

The model has been implemented in FORTRAN by the Naval Surface Warfare Center, Dahlgren, Virginia as part of the Statistical Modeling and Estimation of Reliability Functions for Software (SMERFS). It can be run on an IBM PC (or compatible) or DEC VAX and is available on DOS diskette or magnetic tape, respectively.

Known applications of this model are:

- IBM, Houston, Texas: Reliability prediction and assessment of the on-board NASA Space Shuttle software [SCHN92]
- Naval Surface Warfare Center, Dahlgren, Virginia: Research in reliability prediction and analysis of the TRIDENT I and II Fire Control Software [FARR91]
- NASA JPL, Pasadena, California: Experiments with multi-model software reliability approach [LYU92]
- Hughes Aircraft Co., Fullerton, California: Integrated, multi-model approach to reliability prediction [BOWE87]

6.2.2 Recommended Model: Generalized Exponential Model

6.2.2.1 Generalized Exponential Objectives

Many popular software reliability models yield simular results. The basic idea behind the generalized exponential model is to simplify the modeling process by using a single set of equations to represent models having exponential hazard functions.

The generalized exponential model contains the ideas of several well-known software reliability models. The main idea is that the failure occurence rate is proportional to the number of faults remaining in the software. Furthermore, the failure rate remains constant between failure detections and the rate is reduced by the same amount after each fault is removed from the software. Thus, the correction of each fault has the same effect in reducing the hazard of the software. The objective of this model is to generalize the forms of several well-known models into a form that can be used to forecast:

- Number of failures that will occur by a given time (execution time, labor time, or calendar time)
- Maximum number of failures that will occur over the life of the software
- Maximum number of failures that will occur after a given time
- Time required for a given number of failures to occur
- Number of faults corrected by a given time
- Time required to correct a given number of faults

6.2.2.2 Generalized Exponential Assumptions

The basic assumptions of the Generalized Exponential Model are:

- The failure rate is proportional to the current fault content of a program.
- All failures are equally likely to occur and are independent of each other.
- Each failure is of the same order of severity as any other failure.
- The software is operated in a similar manner as the anticipated operational usage.

- The faults which caused the failure are corrected instantaneously without introduction of new faults into the program.

6.2.2.3 Generalized Exponential Structure

The Generalized Exponentiai Structure begins with a simple, but relatively general, form for the software hazard function:

$$z(x) = K[E_0 - E_c(x)]$$

where

x = a time or resource variable which gauges the progress of the project.

E_0 = the initial number of faults in the program which will lead to failures. It can also be viewed as the number of failures which would be experienced if testing continued indefinitely.

E_c = the number of faults in the program which have been found and corrected once x units of time or effort have been expended

K = a constant of proportionality; failures per resource or time units, per fault remaining

Inspection of this equation shows that the number of remaining faults, E_r, is given by

$$E_r = z(x) / K = [E_0 - E_c(x)]$$

Note that this equation has no fault generation term; it assumes that no new faults which will lead to failures are generated during program debugging. More advanced models that include fault generation are discussed in [MUSA87] and [SHOO83].

Many models in common use can be represented by the above set of equations with various assumptions regarding the

Table 1 Common Reliability Models that Fit the Generalized Exponential Form for the Failure Rate Function

MODEL NAME	ORIGINAL HAZARD FUNCTION	PARAMETER EQUIVALENCES	COMMENTS
Generalized Form	$K[E_0 - E_c(x)]$	-	-
Exponential model [SHOO72]	$K'[E_0 / I_T - \varepsilon_c(x)]$	$\varepsilon_c = E_c / I_T$ $K = K' / I_T$	Normalized with respect to I_T, the number of instructions
Jelinski-Moranda [JELI72]	$\phi[N-(i-1)$	$\phi = K$ $N = E_0$ $E_c = (i-1)$	Applied at the discovery of an error and before it is corrected
Basic Model [MUSA76]	$\lambda_0[1 - \mu / \nu_0]$	$\lambda_0 = KE_0$ $\nu_0 = E_0$ $\mu = E_c$	If the same assumptions are used to predict m and E_c, then this model and the exponential model are the same.
Logarithmic [MUSA83]	$\lambda_0 \exp(-\phi\mu)$	$\lambda_0 = KE_0$ $E_0 - E_c(x)$ $= E_0 \exp(-\phi\mu)$	Basic assumption is that the remaining number of errors decreases exponentially.

parameters and the form that the fault correction function, $E_o(x)$, takes. Some of these models are summarized and compared in Table 1. In the original development of each model in this table, one or more time or resource variables were used. In retrospect, all of the models can, in general, be phrased in terms of any of the time or resource variables given in Table 1. Thus, unless stated to the contrary, the use of a specific time or resource variable does not differentiate one model from another.

Given the data defined in section 6.2.2.5, estimation of any of the model parameters given in Table 1 reduces to a problem in statistical parameter estimation [HOEL71] or [SHOO90a]. There are three basic methods: moments, least squares, and maximum likelihood. Although the original developments of the various models or some of the computer tools available to support these models may have used only one or two of these methods, all three are applicable to each of the models.

The simplest method of parameter estimation is the moment method. Consider the generalized form with its two unknown parameters K and E_o. The classical technique of moment estimation would match the first and second moments of the probability distribution to the corresponding moments of the data. A slight modification of this procedure is to match the first moment, the mean, at two different values of x. That is, letting the total number of runs be n, the number of successful runs be r, the sequence of clock times to failure t_1, t_2, ..., t_{n-r} and the sequence of and the sequence of clock times for runs without failure T_1, T_2, ..., T_r yields,

$$z(x) = \frac{\text{Failures (x)}}{\text{Hours (x)}} = \frac{n\text{-}r}{H} \qquad (6.1)$$

where

$$H = \sum_{i=1}^{n-r} t_i + \sum_{i=1}^{r} T_i$$

Equating the unified form equation with equation (6.1) at two different values of time yields

$$z(x_1) = \frac{n_1 - r_1}{H_1} = K\left[E_o - E_c(x_1)\right] \qquad (6.2)$$

$$z(x_2) = \frac{n_2 - r_2}{H_2} = K\left[E_o - E_c(x_2)\right] \qquad (6.3)$$

Simultaneous solution of these two sets of equations, equations (6.2) and (6.3), yields estimators denoted by ^, for the parameters.

$$\hat{E}_o = \frac{E_c x_1 - \frac{z(x_1)}{z(x_2)} E_c(x_2)}{1 - \frac{z(x_1)}{z(x_2)}}$$

$$= \frac{z(x_2)E_c(x_1) - z(x_1)E_c(x_2)}{z(x_2) - z(x_1)} \qquad (6.4)$$

$$\hat{K} = \frac{z(x_1)}{\hat{F}_o - E_c(x_1)}$$

$$= \frac{z(x_2) - z(x_1)}{E_c(x_1) - E_c(x_2)} \qquad (6.5)$$

Since all of the parameters of the five models in Table 1 are related to Eo and K by simple transformations, equations (6.4) and (6.5) along with the transformations (parameter equivalences) hold. Thus these equations can be used to obtain moment estimates for all the models. For example, we could start using the Musa Basic model of Table 1 and apply the moment estimate procedure to determine $\hat{\lambda}_o$ and $\hat{\nu}_o$ in an analogous fashion to what was done in equations (6.2) and (6.3). More simply, we could use equations (6.4) and (6.5) and the transformation $\nu_o = E_o$ and $\mu_o = KE_o$ to obtain

$$\hat{\nu}_o = \hat{E}_o - \frac{z(x_2)E_c(x_1) - z(x_1)E_c(x_2)}{z(x_2) - z(x_1)}$$

$$\hat{\lambda}_o = \hat{K}\hat{E}_o$$

$$= \frac{z(x_2) - z(x_1)}{E_c(x_1) - E_c(x_2)} * \frac{z(x_2)E_c(x_1) - z(x_1)E_c(x_2)}{z(x_2) - z(x_1)}$$

$$\hat{\lambda}_o = \frac{z(x_2)E_c(x_1) - z(x_1)E_c(x_2)}{E_c(x_1) - E_c(x_2)}$$

Which are the moment estimation equations. Similar results can be obtained for the other models in Table 1. More advanced estimates of the model parameters can be developed using least squares and maximum likelihood estimation theory ([SHOO90a].

6.2.2.4 Generalized Exponential Limitations

The generalized exponential model has the following limitations:

- It does not account for the possibility that each failure may be dependent on others
- It assumes no new faults are introduced in the fault correction process
- Each fault detection may have a different impact on the software when the fault is corrected. The Logarithmic model handles this by saying that earlier fault corrections have a greater impact than later ones.
- It does not account for the possibility that failures can increase over time as the result of program evolution, although techniques for handling this limitation have been developed.

6.2.2.5 Generalized Exponential Data Requirements

During test, a record will be made of each of the total of n test runs. The test results include the r successes and the n-r failures along with the time of occurrence measured in terms of clock time and operational execution time, or test time if operational tests are unavailable. Additionally, there should be a record of the times for the r successful runs. Thus, the desired data is the total number of runs n, the number of successful runs r, the sequence of clock times to failure $t_1, t_2, ..., t_{n-r}$ and the sequence of clock times for runs without failure $T_1, T_2, ..., T_r$. All the times should be for actual or simulated operation; however, if only test time is available, that should be recorded. A description is needed along with the data describing whether is represents operation, simulated operation, or test and the circumstances and conditions governing the input data. If possible, a similar set of operational data should be recorded.

6.2.2.6 Generalized Exponential Applications

The Generalized Exponential Model(s) tend to be optimistic. It is applicable when the operational profile is "regular," and the software debugging process is well controlled (i.e., the fault correction process tends to be complete and not error prone.)

The major model applications are described below. These are separate but related uses of the model that, in total, comprise an integrated reliability program.

- Forecasting: forecasting future failures, fault corrections, and related quantities described in section 6.2.2.7.
- Control: comparing forecast results with predefined goals and flagging software that fails to meet those goals.
- Assessment: determining what action to take for software that fails to meet goals (e.g., intensify inspection, intensify testing, redesign software, revise process). The formulation of test strategies is also part of the assessment. Test strategy formulation involves the determination of: priority, duration, and completion date of testing, allocation of personnel, and allocation of computer resources to testing.

6.2.2.7 Reliability Forecasts

Besides the estimate of the total number of faults given by $\hat{E}_o$, other estimates are:

- Estimated time to remove the next m faults

$$= \sum_{j=n+1}^{n+m} \frac{1}{\hat{K}_0\left(\hat{E}_0 - j + 1\right)}$$

- Estimate of the current failure rate at time

$$\tau = \hat{K}_0\left(\hat{E}_0 \exp\left(-\hat{K}_0\tau\right)\right)$$

For other quantities that can be estimated, see the references listed in paragraph 6.2.2.8.

6.2.2.8 Generalized Exponenential Implementation Status and Reference Applications

The Generalized Exponential Model has not been implemented as a standalone model. The many models it represents, however, have been implemented in several tools including SMERFS from the Naval Surface Warfare Center, Dahlgren, VA, Software Reliability Modeling Program (SRMP) from the the Center for Software Reliability in London, England, and RELTOOLS from AT&T. See Appendix B for details.

While the generalized exponential model has not been used widely, many of the specific models that it covers as special cases have been applied successfully. See the following for example applications:

- Jelinski Z.and Moranda, P. B., "Software Reliability Research," W. Freiberger, Editor, *Statistical Computer Performance Evaluation*, Academic Press, New York, pp. 465-484.

- Shooman, M. L. and Richeson, G., "Reliability of Shuttle Control Center Software," Proceedings Annal Reliability and Maintainabilty Symposium, January 1983, pp. 125-135.

- Kruger, G. A., "Validation and Further Application of Software Reliability Growth Models," Hewlett-Packard Journal, April 1989, pp. 75-79.

6.2.3 Recommended Model: Musa / Okumoto Logarithmic Poisson Execution Time Model

6.2.3.1 Musa / Okumoto Objectives

The logarithmic Poisson is especially applicable when the testing is done according to an operational profile that is very nonuniform in nature. Early fault corrections have a larger impact on the failure intensity function than later ones. The failure intensity function tends to be convex with decreasing slope for this situation. Thus a logarithmic Poisson model may be very appropriate for this circumstance.

If one is also interested in relating calendar time considerations (e.g., completion of testing, resource management, etc.) to reliability, the logarithmic Poisson is the only non-exponential model that can do this at this time.

Considerations relating to computer utilization, personnel level, and current and projected failure rate trade-offs can be performed to balance reliability considerations with time and resource constraints.

The number of failures occurring over an infinite amount of time is unbounded for this model [MUSA87]. It is especially applicable when high nonuniformity is experienced in the operational profile. The belief is that as one detects the earlier faults a greater reduction in the failure intensity is experienced. With a highly non-uniform profile exhibited, early fault corrections make a more substantial impact on the failure behavior of the software than later ones. This behavior of the failure intensity can be more adequately modeled by a logarithmic Poisson approach.

If there is a decreasing effectiveness of the repair process, then this model can yield an unbounded number of failures even though the number of faults may be finite.

6.2.3.2 Musa / Okumoto Assumptions

The specific assumptions for this model are:

- The software is operated in a similar

manner as the anticipated operational usage.

- Failures are independent of each other.
- The failure intensity decreases exponentially with the expected failures experienced.

Note: There are two consequences of the third assumption. First, the expected number of failures is a logarithmic function of time. Second, the model may report an infinite number of failures.

6.2.3.3 Musa / Okumoto Structure

From the model assumptions we have:

$\lambda(\tau) =$ failure rate function after t amount of execution time has been expended

$$= \lambda_o \exp[-\theta\mu(\tau)]$$

The parameter λ_o is the initial failure rate function and θ is the failure rate decay parameter with $\theta > 0$.

Using a reparameterization of $\beta_o = \theta^{-1}$ and $\beta_1 = \lambda_o\theta$, then the maximum likelihood estimates of β_o and β_1 are shown in [MUSA87] to be the solutions of the following equations:

$$\theta = \hat{\beta}_o - \frac{n}{\ln\left(1+\hat{\beta}_1 t_n\right)}$$

$$\theta = \frac{1}{\hat{\beta}_1}\sum_{i=1}^{n}\frac{1}{1+\hat{\beta}_1 t_n} - \frac{nt_n}{\left(1+\hat{\beta}_1 t_n\right)\ln\left(1+\hat{\beta}_1 t_n\right)}$$

Here t_n is the cumulative CPU time from start to the current time. Over this period, we have observed a total of n failures. Once maximum likelihood estimates are found for β_o and β_1, the maximum likelihood estimates for θ and λ_o are, using the invariance property of such estimators:

$$\hat{\theta} = \frac{1}{n}\ln\left(1+\hat{\beta}_1 t_n\right) \text{ and}$$

$$\hat{\lambda}_o = \hat{\beta}_o\hat{\beta}_1$$

6.2.3.4 Musa / Okumoto Limitations

Two limitations are:

- The failures may not be independent of one another.
- The failure intensity may rise as modifications are made to the software.

6.2.3.5 Musa / Okumoto Data Requirements

The required data is either:

- The time between failures, i.e., the X_i's.
- The time of the failure occurrences, i.e.,

$$t_i = \sum_{j=1}^{i} X_j$$

6.2.3.6 Musa / Okumoto Applications

The major model applications are described below. These are separate but related applications that, in total, comprise an integrated reliability program.

- Prediction: Estimating future failure times, fault corrections, and related quantities described in Musa's book [MUSA87].
- Control: Comparing prediction results with pre-defined goals and flagging software that fails to meet goals.
- Assessment: Determining what action to take for software that fails to meet goals (e.g., intensify inspection, intensify testing, redesign software, revise process). The formulation of test strategies is also a part of assessment. It involves the determination of: priority, duration and completion date of testing, and allocation of personnel and computer resources to testing.

6.2.3.7 Reliability Forecasts

In their book, Musa, Iannino, and Okumoto [MUSA87] show that from the assumptions above and the fact that the derivative of the mean value function is the failure rate function, we have:

$$\hat{\lambda}(\tau) = \frac{\hat{\lambda}_o}{\hat{\lambda}_o \theta \tau + 1}$$

$\hat{\mu}(\tau)$= mean number of failures experienced by the time τ is expended

$$= \frac{1}{\hat{\theta}} \ln\left(\hat{\lambda}_o \hat{\theta} \tau + 1\right)$$

The estimates of additional reliability measures are provided in the references listed in paragraph 6.2.3.8.

6.2.3.8 Musa / Okumoto Implementation Status and Reference Applications

The model has been implemented by the Naval Surface Warfare Center, Dahlgren, VA as part of SMERFS. It can be run on any computer system with a FORTRAN compiler and is available upon request.

This model has also been implemented in the set of programs written by AT&T (see Appendix B for details).

This model has been applied widely. See the following for example applications:

- Musa, J. D., Iannino, A., and Okumoto, K., *Software Reliability: Measurement, Prediction, and Application*, New York, McGraw-Hill, 1987.

- Musa, J. D. and Okumoto, K., "A Logarithmic Poisson Execution Time Model for Software Reliability Measurement," Proceedings of the 7th International Conference on Software Engineering, Orlando, FL, 1984, pp. 230-238.

- Ehrlich, W. K., Stampfel, J. P., and Wu, J. R., "Application of Software Reliability Modeling to Product Quality and Test Process," Proceedings of the IEEE/TCSE Subcommittee on Software Reliability Engineering Kickoff Meeting, NASA Headquarters, Washington, DC, April 1990, paper 13.

6.2.4 Recommended Model: Littlewood / Verrall Model

6.2.4.1 Littlewood / Verrall Objectives

The intention of the Littlewood / Verrall is to model the doubly stochastic nature of the software failure process. There are two basic sources of uncertainty which need to be taken into account when software fails and fixes are attempted.

In the first place there is uncertainty about the nature of the operational environment: we do not know when a certain input will show itself, and in particular we do not know which inputs will be selected next. Thus, even if we had complete knowledge of which inputs were failure-prone (and of course this is never the case), we still could not tell with certainty when the next one to induce a failure would be received. All software reliability models recognize this source of uncertainty, and it is often presented mathematically by a simple Poisson process: i.e., it is assumed that failures occur *purely randomly*. This means the time to next failure, for example, will have an exponential distribution.

The second source of uncertainty concerns what happens when an attempt is made to remove the fault that caused the failure. The aforementioned models that assume that the process of failures is locally purely random, it is this uncertainty that governs the changes in the failure rate as debugging proceeds: i.e., it determines the nature of the *reliability growth*. There is uncertainty here for two main reasons. In the first place, it is clear that not all the faults contribute the same amount to the unreliability of a program. Some contribute a greater amount than others. If the software has failed because a fault has been detected that contributes a large amount to the overall reliability, then there will be a correspondingly large increase in the

reliability (reduction in the failure rate) when this is removed. In the second place, we can never be sure that we actually have removed a fault successfully; indeed it is possible that some new fault has been introduced and the reliability of the program made worse. The result of these two effects is that the failure rate of a program changes in a random way as debugging proceeds: there will likely be a downwards jump in this rate at each fix attempt, but this is not certain, and the size of the jump in unpredictable.

The Littlewood / Verrall model, unlike the other models discussed, takes account of both of these sources of uncertainty in the failure process - that due to basic unpredictability of the environment which profers inputs for execution, and that due to an intrinsic uncertainty of the effects of the human activities during debugging.

6.2.4.2 Littlewood / Verrall Assumptions

The following assumptions apply to the Littlewood / Verrall model:

- The software is operated during the collection of failure data in a manner that is similar to that for which predictions are to be made; the test environment is an accurate representation of the operational environment.
- The times between successive failures are conditionally independent exponential random variables, i.e., locally (between failures) the failure process is purely random.
- The fixing process involves uncertainty represented by allowing the successive failure rates, following successive fix attempts, to be a sequence of independent random variables.

6.2.4.3 Littlewood / Verrall Structure

This model treats the successive rates of occurrence of failures as fixes take place, as random variables. It assumes

$$P(t_i|\Lambda_i = \lambda_i) = \lambda_i e^{-\lambda_i t_i}$$

The sequence of rates λ_i is treated as a sequence of independent stochastically decreasing random variables. This reflects the likelihood, but not certainty, that a fix will be effective. It is assumed that

$$g(\lambda_i) = \frac{\psi(i)^{\alpha}\lambda_i^{\alpha-1}e^{-\psi(i)\lambda_i}}{\Gamma(\alpha)} \text{ for } \lambda_i > 0$$

which is a gamma distribution with parameters $\alpha, \psi(i)$.

The function $\psi(i)$ determines the reliability growth. If, as is usually the case, $\psi(i)$ is an increasing function of i, it is easy to show that Λ_i forms a stochastically decreasing sequence. For this model a fix may make the program less reliable, and even if an improvement takes place it is of uncertain magnitude.

By setting $\psi(i)$ to either $\beta_o + \beta_1 i$ or $\beta_o + \beta_1 i^2$ and eliminating α, Littlewood and Verrall present a method of estimating β_o and β_1 based upon maximum likelihood. By eliminating α from the likelihood equations; i.e., the estimate of a can be expressed as a function of the estimates of the other two parameters. See [FARR83,LITT73] for details. The maximum likelihood calculation needs to be done using a numerical optimization routine which is available in commercially available software, such as those found in Appendix B.

Least squares estimates of the parameters $(\alpha, \beta_o, \beta_1)$ are found by minimizing:

$$S(\alpha,\beta_o,\beta_1) = \sum_{i=1}^{n}\left(x_i - \frac{[\psi(i)]}{\alpha - 1}\right)^2$$

See [FARR83] for further details.

6.2.4.4 Littlewood / Verrall Limitations

The primary limitation as with all Bayesian analysis is the specification of the prior

density function $g(\lambda_i)$. A secondary limitation of the Littlewood / Verrall model is that it cannot estimate the number of faults remaining in the software (the estimate may be infinite depending on the $\psi(i)$ function).

6.2.4.5 Littlewood / Verrall Data Requirements

The only required data is either:

- The time between failures, i.e. the X_i's.
- The time of the failure occurrences, i.e.

$$t_i = \sum_{j=1}^{i} X_i$$

6.2.4.6 Littlewood / Verrall Applications

The Littlewood / Verrall Model (or Inverse Polynomial Model) is a conservative and pessimistic model. It is applicable when the operational profile is non-uniform and even irregular, especially when the software debuging process is imperfect (i.e., the fault correction process tends to be incomplete or error-prone). This model has the capability of adjusting the parameters to reflect the situation.

The major model applications are described below. These are separate but related uses of the model that, in total, comprise an integrated reliability program.

- Forecasting: Forecasting future failures, fault corrections, and related quantities described in section 6.2.1.7.
- Control: Comparing forecast results with pre-defined goals and flagging software that fails to meet those goals.
- Assessment: Determining what action to take for software that fails to meet goals (e.g., intensify inspection, intensify testing, redesign software, revise process). The formulation of test strategies is also part of assessment. Test strategy formulation involved the determination of: priority, duration and completion date of testing, allocation of personnel, and allocation of computer resources to testing.

6.2.4.7 Reliability Forecasts

Estimation of reliability and other associated terms is via substitution of the parameter estimates into appropriate expressions. An estimate of the Mean Time To Failure, (MTTF), is:

$$\hat{MTTF} = \hat{E}(X_i) = \frac{[\hat{\psi}(i)]}{\hat{\alpha} - 1}$$

The expression for failure rate is:

$$\hat{\lambda}(t) = \frac{\hat{\alpha}}{(t + \hat{\psi}(i))}$$

(Note that the failure rate expression is a continuously decreasing function during periods of failure-free working, representing the greater confidence that comes from such evidence)

The reliability function is:

$$R(t) = P(T_i > t) = \hat{\psi}(i)^{\hat{\alpha}}[t + \hat{\psi}(i)]^{-\hat{\alpha}}$$

In all of the above expressions, $\hat{\psi}(i)$ and $\hat{\alpha}$ are the estimates of the two respective parameters from section 6.2.4.3.

For other quantities that can be estimated, see the references listed in paragraph 6.2.4.8.

6.2.4.8 Littlewood / Verrall Implementation Status and Reference Applications

The model has been implemented as part of the SMERFS. It can be run on any computer system with a FORTRAN compiler and is available upon request.

This model has also been implemented in the Software Reliability Modeling Programs (SRMP) at the Center for Software Reliability in London, England by Dr. Littlewood and his associates of Reliability and Statistical

Consultants, Ltd. This program package runs in a PC environment.

The Littlewood/Verrall model has been applied widely. See the following for examples of applications:

- K. Kanoun, J. Sabourin, (1987), "Software Dependability of a Telephone Switching System," Proceedings 17th IEEE Symposium on Fault-Tolerant Computing (FTCS-17), Pittsburgh, PA.

- Mellor, P., (1986), "State of the Art Report on Software Reliability," *Infotech*, London

- Abdel-Ghaly, A. A., Chan, P. Y. and Littlewood, B., (1986), "Evaluation of Competing Software Reliability Predictions," *IEEE Transactions of Software Engineering*, SE-12 (9), 950-967

6.3 Experimental Approaches

Several improvements to the software reliability models described in the previous sections have been recently proposed. First, researchers at the City University of London have devised a method of recalibrating the models [BROC92] to reduce their biases (see section 6.1.1.2). These findings to date suggest that the recalibrated models yield consistently more accurate forecasts than the uncalibrated models. Second, work has also been done in combining the results from two or more models in a linear fashion to increase predictive accuracy [LYU92, LU92]. This work suggests that such combinations yield more accurate results than individual models. The advantage of combining model results is the simplicity with which the combinations are formed – the models in the combination are executed individually, with only the results being combined. Third, efforts to incorporate software complexity metrics into reliability models [KAFU87, KHOS91], and to gauge the effects of different types of testing (e.g., branch testing, data path testing) on reliability growth [MATH92] are being investigated. Finally, the use of neural networks for software reliability parameter estimation is being investigated [KARU92].

Although these efforts show promise in increasing the forecasting accuracy of software reliability modeling, there is not sufficient evidence to classify them as recommended practice at this time. They are included here to indicate some of the current avenues of investigation. Further experience with these methods may lead to their being classified as recommended practice in the future.

7.0 SOFTWARE RELIABILITY DATA

A variety of applications for software reliability measurement were described in Section 5 of this document. Section 6 provided a list of selection criteria as well as a set of models for estimating the reliability of the software product. Data collection provides the foundation on which both of these sections depend. This section addresses (1) a procedure for collecting data, (2) two data types, (3) the relationships between the two types, and (4) the AIAA data base hierachy.

7.1 Data Collection Procedure

The following nine steps can be used to establish a software reliability data collection process:

- Step 1: Establish the objectives.

The first step in planning to collect data is to determine the objectives of the data and what data items will be collected. Data collection does involve cost, so each item should be examined to see if the need is worth the cost. This should be done in the context of the planned application or applications of software reliability engineering. If the item is questionable, consider alternatives such as approximating the item or collecting it at a lower frequency. Look for possibilities of collecting data items that can serve multiple purposes. If this careful examination is not performed, the unnecessary burden in effort and cost on the project can result in the degradation of all data or even the abandonment of the effort.

- Step 2: Plan the data collection process.

It is recommended that all parties (designers, coders, testers, users, and key management) participate in the planning effort. The data collectors must be motivated if quality data is to be collected. Present the goals of the data collection effort. Relate it to their direct personal benefit. This will insure that all parties understand what is being done and the impact it will have on their respective organizations.

It is suggested that a first draft data collection plan be presented as a starting point. The plan should include topics such as:

- What data items will be gathered?
- Who will gather the data?
- How often will the data be reported?
- Formats for data reporting (e.g., electronic spreadsheet, and paper forms)
- How is the data to be stored and processed?
- How will the collection process be monitored to ensure integrity of the data?

Solicit identification of problems with the plan and desired improvements. Elicit the participation of the data collectors in the solution of any problems. It will provide them an opportunity to provide new ideas and insight into the development process. Support will be gained by having the parties that will be affected as active participants.

Recording procedures should be carefully considered to make them as simple as possible. Solicitation of data from project members can reduce effort and make collection more reliable.

For the failure count method, the data collection interval should be selected to correspond to the normal reporting interval of the project from which data are being collected (e.g., week, month) or an integral multiple thereof. This will facilitate obtaining data on the level of effort devoted to the software under test (person-hours and computer hours) which must be correlated with the reliability data.

- Step 3: Apply tools.

Availability of tools identified in the collection process must be considered. If the tools are not commercially available then time needs to be planned for their development. Furthermore, the amount of automatic data collection must be considered. To minimize the impact on the project's schedule, automated tools should be considered whenever possible.

When decisions are being made to automate the data collection process for either of the two types of data one needs to weigh certain factors. These include:

- Availability of the tool. Can it be purchased or must it be developed?
- What is the cost involved in either the purchase of the tool or its development?
- When will the tool be available? If it must be developed, will its development schedule coincide with the planned use?
- What impact will the data collection process have on the development schedule?
- Can the tool handle adjustments that may be needed? Can the adjustments be completed in a timely manner?
- How much overhead (people and computer time) will be needed to keep the data collection process going?

Once the tool has been developed and implemented, one needs to consider ways of ensuring the right data are being gathered. Flexibility also should be designed into the tool, as data collection requirements may change. Finally, one needs to make some type of assessment of not only what the tool saved in time and resources but also what the data collection process gained. Records could be kept of the number of faults detected after the release of the software. This could be compared with reliability estimates of similar projects that did not employ this methodology. Estimates of reduced maintenance and fault correction time could be made based upon the estimated current failure rate.

For the tool itself, one could estimate the amount of time and effort that would be expended if the data had been collected manually. These statistics could then yield cost estimates which would be compared with the procurement and implementation costs of the automated tool. If the cost of the automated tool is significantly higher, one certainly would question the wisdom of developing the tool. However, even if the costs come out higher, consideration must be given to future use of the tool. Once the tool has been developed it may be easily adapted over many software development efforts and could yield significant savings.

• Step 4: Provide training.

Once the tools and plans are in place, training of all concerned parties is important. The data collectors need to understand the purpose of the measurements and know explicitly what data are to be gathered.

• Step 5: Perform trial run.

A trial run of the data plan should be made to resolve any problems or misconceptions about the plan. This can save vast amount of time and effort when the "real thing" occurs.

• Step 6: Implement the plan.

Data must be collected and reviewed promptly. If this is not done, quality will suffer. Generate reports to show project members; they can often spot unlikely results and thus identify problems. Problems should be resolved quickly before the information required to resolve them disappears.

• Step 7: Monitor data collection.

Monitor the process as it proceeds to insure the objectives are met and the program is meeting its established reliability goals.

• Step 8: Use the data.

Don't wait to the end after the software has been released to the users to make your reliability assessments. Estimating software reliability at regular, frequent intervals will maximize visibility into the development effort, permitting managerial decisions to be made on a regular basis.

• Step 9: Provide feedback.

This should be done as early as possible during the data collection. It is especially important to do so at the end. Those who were involved want to hear what impact their efforts had. If no feedback is given, you'll find yourself facing the problem alluded to in the beginning of this section. Namely, the parties will resist further future efforts because they see no purpose. Again, why collect data for the sake of collecting it?

7.2 Failure Count Data vs Execution Time Data

It is generally accepted that execution (CPU) time is superior to calendar time for software reliability measurement and modeling. If execution time is not readily available, approximations such as clock time, weighted clock time, or units that are naturals to the applications, such as transactions, may be used [MUSA87, pp 156-158].

The following paragraphs address failure-count and execution time data collection to support the recommended models identified in Section 6.

7.2.1 Failure-Count Data

Since the recommended models employ the number of failures detected per unit of time, these data are usually readily available. Most organizations have some type of configuration management process in place. As part of this process, a procedure for reporting failures and approving changes to the software is in place. The software problem reporting mechanism may be either manual or automatic. In addition, the problem reports may be stored within a computer data base system or a manual filing system. The key is that the data can be easily extracted.

Make sure that the problems are really software problems - some organizations use problem reporting for any type of anomaly and the time recorded on a problem report may not be the time at which the failure was experienced, it may be the time in which the

report was filled out.

Another pitfall to avoid when using problem reporting data involves forming the time intervals. Remember, the purpose is to model the number of failures detected per unit of time within a specified environment. These units should therefore be consistent in duration, manpower, and testing intensity.

Usually the information to check this is not available. All one has is data on the number of failures detected in one period or another. However, there may have been twice as many testing personnel in one period than the other. The only way to find out this information is to seek it out. This may involve talking with the testers or even reviewing old time sheets covering the period of interest. Generally, the longer the period of time in which the fault counts are formed the more smoothing occurs. Variations within short intervals of time will be averaged out over the longer time units.

Data may be gathered at any point within the development cycle beginning with the system test phase. Overall measurement objectives will help you determine the rate (failures reported per week, per month, or per quarter) at which data is collected. It is suggested that you start out using the number of failures reported over the shortest unit of time consistent with your objectives. If good fits are not achieved, combine intervals to the next level. For example: days to weeks, or weeks to quarters. The smoothing effect mentioned in the previous paragraph may help in the modeling process.

7.2.2 Execution Time Data

This data may be collected directly or indirectly. Also, it is best to collect, when feasible, the actual execution time of a program rather than the amount of wall clock time or system active time expended. This is the actual amount of time spent by the processor in executing the instructions. Execution time gives a truer picture of the stress placed on the software. You could have large amounts of time expended on the clock but very little computations may have to be done during this period. This yields small execution times. This would tend to give overly optimistic views of the reliability of the software. Modeling using execution time data tends to give superior results than simple elapsed wall clock time or system active time. However, the data may be difficult to collect since a monitor of the actual operating system is involved. Another source for obtaining this data is to adjust the wall clock time by a factor that represents the average computer utilization per unit of wall clock time.

If the time-between failures (wall clock or execution time) is unavailable and only grouped data (number of failures occurring per unit of time) is available, the time-between-failures can still be obtained. One way is to randomly allocate the failures over the length of the time interval. Randomization will not cause errors in estimation for some of the models by more than 15 percent [MUSA87, pg.128]). A second way is the easiest to implement. Simply allocate the failures uniformly over the interval length. For example, suppose the interval is three hours in duration and three failures occurred during this period. We could then treat the time-between-failures to be each one hour in length.

Two additional considerations are: (1) adjusting the failure times to reflect an evolving program and (2) handling multiple sites / versions of the software. In the first situation, the failure intensity may be underestimated in the early stages of the program's development yielding overly optimistic views of the reliability. For the second consideration, there are multiple versions of the code being executed at different locations. In [MUSA87, pp. 162-176] both considerations are addressed.

7.3 Transformations Between the Two Types of Input

Programs may have the capability to estimate model parameters from either failure-count or time-between-failures data, as maximum likelihood estimation can be applied to both. However, if a program accommodates only one type of data, it is easy to transform to the other type.

If the expected input is failure-count data, it

may be obtained by transforming time-between-failures data to cumulative time data and then simply counting the cumulative times that occur within a specified time period.

If the expected input is time-between-failures data, convert the failure-count data by randomly selecting a number of cumulative failure times in the period equal to the count and then finding the time differences between them [MUSA87, pp. 143-146].

7.4 The AIAA Repository

The AIAA sponsored the development of a software reliability project repository. This repository contains data for both researchers and practioners alike.

7.4.1 Minimum Data Required

The following information represents a minimum subset of data that should be collected for any software project. It will be found useful in developing and maintaining local organization repositories as well.

I. Project Data

The data should contain information to identify and characterize each system and effort that generates data stored in the database. Project data should allow users to categorize projects based on application type, development methodology and environment, scale, required reliability or currency. The following project-related data are suggested:.

- The name of each life-cycle activity (e.g., requirements definition, design, code, test, operations)
- The start and end date for each life-cycle activity
- The effort spent (in staff months) during each life-cycle activity[1]
- Characterize the development environment (organic, semi-detached, or embedded)[2]

II. Component Data

For each system component (e.g., subsystem, element, or module) provide the following:

- Software size in terms of executable source lines of code as well as the number of comments and the total number of object instructions
- The source language used

III. Dynamic Failure Data

For each failure recorded the following information should be tracked:

- The activity being performed when the problem was detected (e.g., testing, operations, and maintenance)
- The date and time of the failure
- The severity of the failure (e.g., critical, major, minor)

And at least one of the following data items:

- The number of CPU hours since the last failure
- The number of runs or test cases executed since the last failure
- The number of wall clock hours since the last failure
- The number of test hours per test interval and number of failures detected in the interval
- Test labor hours since the last failure

IV. Fault Correction Data

For each failure corrected with a software fix, the following information should be recorded:

---

[1] primarily required of resource modeling.

[2] as referenced in the COCOMO framework[BOEH81]

- The date and time the fix was available
- The labor hours required for correction

Also record at least one of the following data items consistent with the selected data item from the dynamic failure data list.

- The CPU hours required for the fix
- The number of runs required to make the fix
- The wall clock hours used to make the correction

Finally, it is important to maintain corporate knowledge of the software testing and debugging effort. Therefore, have a point of contact who knows the project write down the lessons learned and have that person available to answer questions concerning the data (how they were obtained and how some of the project specific terminology translates to the current terminology).

7.4.2 Input for Practitioners

The above data are for use by practitioners who are interested in finding projects similar to their own projects. It also provides a guideline for defining data collection requirements when new projects are started.

7.4.3 Input for Researchers

In addition to the minimum data mentioned in Section 7.4.1, the AIAA Repository also contains data for research studies in software reliability measurement. These data items include:

I. Project Data

- Remarks about the development schedule (e.g., replans, problems, corrective actions)
- The average staff size (in staff hours) and development team experience (in years)
- The most important requirements, design, code, test, and configuration management tools and / or methods used
- The number of different organizations developing software for the project
- The Software Engineering Institute (SEI) index of the development environment and the assessment method
- The most important tool and model used for software reliability estimation

II. Component Data

- The name and model of the development and target hardware
- Average and peak computer resource utilization (e.g., CPU busy, memory utilization, and input / output channel utilization)

III. Dynamic Failure Data

- The type of the failure (e.g., interface, syntax)
- The method of fault / failure detection (e.g., inspection, system abort, invalid output)
- The unit complexity (e.g., McCabe Cyclomatic) and size where the fault was detected

IV. Fault Correction Data

- The type of fix (e.g., software change, documentation change, requirements change, no change)

8.0 BIBLIOGRAPHY

1. [ABDE86] Abdel-Ghaly, A. A., Chan, P. Y., and Littlewood, B., "Evaluation of Competing Software Reliability Predictions," *IEEE Transactions on Software Engineering*, SE-12, 9, pp 950-967, 1986.

2. [BOEH81] Boehm, B. W., *Software Engineering Economics,* Prentice-Hall, New York, 1981.

3. [BOWE87] Bowen, John B., "Application of a Multi-Model Aproach

to Estimating Residual Software Faults and Time Between Failures," *Quality and Reliability Engineering International*, Vol. 3. 41-51 (1987) pp. 41-51.

4. [BROC92] Brocklehurst, S. and Littlewood, B., "New Ways to Get Reliability Measures," *IEEE Software*, July 1992, pp. 34-42.

5. [BROO80] Brooks, W.D. and Motley, R.W., *Analysis of Discrete Software Reliability Models*, Technical Report #RADC-TR-80-84, Rome Air Development Center, 1980

6. [CROW77] Crow, L., *Confidence Interval Procedures for Reliability Growth Analysis*, Technical Report #197, U.S. Army Material Systems Analysis Activity, Aberdeen Proving Grounds, Maryland.

7. [DUAN64] Duane, J.T., "Learning Curve Approach to Reliability Monitoring," *IEEE Transactions on Aerospace*, Volume 2, pp. 563-566.

8. [FARR83] Farr, W. H., *A Survey of Software Reliability Modeling and Estimation*, Technical Report #82-171, Naval Surface Warfare Center, Dahlgren, Virginia.

9. [FARR91] Farr, W. H. and Smith, O. D., *Statistical Modeling and Estimation of Reliability Functions for Software (SMERFS) Users Guide*, NAVSWC TR-84-373, Revision 2, Naval Surface Warfare Center, Dahlgren, Virginia.

10. [FREE88] Freedman, R. S. and M. L. Shooman, *An Expert System for Software Component Testing*, Final Report, New York State Research and Development Grant Program, Contract No. SSF(87)-18, Polytechnic University, Oct. 1988.

11. [GIFF84] Gifford, D., and Spector, A., "The TWA Reservation System," *Communications of the ACM*, pp. 650-665, Vol 2, No. 27, July 1984.

12. [GOEL79] Goel, A. and Okumoto, K., "Time-Dependent Error-Detection Rate for Software Reliability and Other Performance Measures," *IEEE Transactions on Reliability*, Vol. R-28, No. 3, pp. 206-211.

13. [HECH86a] Hecht, H. and Hecht, M., "Software Reliability in the System Context," *IEEE Transactions on Software Engineering*, January 1986.

14. [HECH86b] Hecht, H. and Hecht, M., "Fault Tolerant Software," in *Fault Tolerant Computing*, D. K. Pradhan, ed., Prentice Hall, 1986.

15. [HECH89] Hecht, H., "Talk on Software Reliability," given at AIAA Software Reliability Committee Meeting, Colorado Springs, CO., August 22-25, 1989.

16. [HOEL71] Hoel, P. G., *Introduction to Mathematical Statistics*, Fourth Edition, John Wiley & Sons, New York, NY, 1971.

17. [IYER83] Iyer, R. K., and Velardi, P., *A Statistical Study of Hardware Related Software Errors in MVS*, Stanford University Center for Reliable Computing, October 1983.

18. [JELI72] Jelinski, Z. and P. Moranda, "Software Reliability Research," in W. Freiberger, ed., *Statistical Computer Performance Evaluation*, Academic Press, New York, NY, 1972, pp. 465-484.

19. [JOE85] Joe, H. and Reid, N., "Estimating the Number of Faults in a System," *Journal of the American Statistical Association*, 80(389), pp. 222-226.

20. [KAFU87] Kafura, D. and Yerneni, A., *Reliability Modeling Using Complexity Metrics*, Virginia Tech University Technical Report, Blacksburg, VA, 1987

21. [KANO87] Kanoun, K. and Sabourin,

T., "Software Dependability of a Telephone Switching System," *Proc. 17th IEEE Int. Symposium on Fault-Tolerant Computing* (FTCS-17), Pittsburgh, PA, 1987.

22. [KARU92] Karunantithi, N., Whitely, D., and Malaiya, Y. K., "Using Neural Networks in Reliability Prediction," *IEEE Software,* July 1992, pp. 53-60.

23. [KHOS91] Munson, J. C. and Khoshgoftaar, T. M., "The Use of Software Complexity Metrics in Software Reliability Modeling," *Proceedings of the International Symposium on Software Reliability Engineering*, Austin, TX, May 1991, pp. 2-11.

24. [KLIN80] Kline, M. B., "Software & Hardware R&M: What are the Differences?" *Proceedings Annual Reliability and Maintainability Symposium*, 1980, pp. 179-185.

25. [LAPR84] Laprie, J. C., "Dependability Evaluation of Software Systems in Operation," *IEEE Trans. on Software Eng.*, Vol. SE-10, Nov 84, pp 701-714.

26. [LIPO86] Lipow, M. and Shooman, M. L., "Software Reliability," in Consolidated Lecture Notes Tutorial Sessions Topics in Reliability & Maintainability & Statistics, Annual Reliability and Maintainability Symposium, 1986 .

27. [LITT73] Littlewood, B. and Verrall, J. L., (June 1974, "A Bayesian Reliability Model with a Stochastically Monotone Failure Rate," *IEEE Transactions on Reliability*, pp. 108-114.

28. [LITT79] Littlewood, B. "Software Reliability Model for Modular Program Structure," *IEEE Trans. on Reliability*, R-28, pp. 241-246, Aug. 1979.

29. [LITT86] Littlewood, B., Ghaly, A., and Chan, P. Y., "Tools for the Analysis of the Accuracy of Software Reliability Predictions", (Skwirzynski, J. K., Editor), *Software System Design Methods,* NATO ASI Series, F22, Springer-Verlag, Heidleberg, pp. 299-335.

30. [LITT90] Fenton, N. and Littlewood, B., "Limits to Evaluation of Software Dependability," *Software Reliability and Metrics*, Elsevier Applied Science, London, pp. 81-110.

31. [LLOY77] Lloyd, D. K. and Lipow, M. *Reliability: Management, Methods, and Mathematics,* 2nd Edition, 1977, ASQC.

32. [LU92] Lu, M., Brocklehurst, S., nd Littlewood, B., "Combination of Predictions Obtained from Different Software Reliability Growth Models," *Proceedings of the Tenth Annual Software Reliability Symposium*, Denver, CO, June 1992, pp. 24-33.

33. [LYU92] Lyu, M. R. and Nikora, A., "Applying Reliability Models More Effectively," *IEEE Software*, July 1992, pp. 43-52.

34. [MATH92] Mathur, A. P., and Horgan J. R., "Experience in Using Three Testing Tools for Research and Education in Software Engineering," *Proceedings of the Symposium on Assessment of Quality Software Development Tools*, pp. 128-143, May 27-29, 1992, New Orleans, LA

35. [MCCA87] McCall, J. A., et al, "Methodology for Software and System Reliability Prediction," Final Technical Report, Prepared for RADC, Science Applications International Corporation, November 1987, RADC-TR-87-171.

36. [MELL86] Mellor, P., "State of the Art Report on Software Reliability." *Infotech*, London, 1986.

37. [MIL-HDBK-217E] Military Handbook Reliability Prediction of Electronic Equipment, MIL-HDBK-217E, Rome Air Development Center, Griffis AFB, NY 13441-5700, Oct. 27, 1986. Naval Publications and Forms Center, Code 3015, 5801 Tabor Ave., Philadelphia, PA 19120.

38. [MUSA75] Musa, J., "A Theory of Software Reliability and Its Application," *IEEE Trans. Software Eng.*, Vol. SE-1, No. 3, September 1975, pp 312-327.

39. [MUSA84] Musa, J.D., Okumoto, K., "A Logarithmic Poisson Execution Time Model for Software Reliability Measurement," *Proceedings Seventh International Conference on Software Engineering*, Orlando, pp. 230-238.

40. [MUSA87] Musa, J. D., Iannino, A., and Okumoto, K., *Software Reliability: Measurement, Prediction, Application.* McGraw-Hill, New York, 1987.

41. [MUSA92] Musa, J. D., "Determining the Operational Profile," available from the author.

42. [NPRD85] Nonelectronic Parts Reliability Data, NPRD-3, Reliability Analysis Center, Rome Air Development Center, Griffis AFB, NY 13441-5700, 1985, NTIS ADA163514

43. [ROOK91] Rook, P. *Software Reliability Handbook,* Elsevier Applied Science, London, 1990.

44. [SCHN75] Schneidewind, N. F., "Analysis of Error Processes in Computer Software," Proceedings of the International Conference on Reliable Software, IEEE Computer Society, 21-23 April 1975, pp. 337-346.

45. [SCHN92] Schneidewind N. F. and Keller, T. M., "Applying Reliability Models to the Space Shuttle," *IEEE Software*, July 1992, pp. 28 - 33.

46. [SHOO76] Shooman, M. L., "Structural Models for Software Reliability Prediction," Second National Conf. on Software Reliability, San Francisco, CA, October 1976.

47. [SHOO83] Shooman, M. L., *Software Engineering: Design, Reliability, and Management,* McGraw-Hill Book Co, New York, NY, 1983.

48. [SHOO90a] Shooman, M. L., *Probabilistic Reliability: An Engineering Approach*, McGraw-Hill Book Co., New York, NY, 1968, 2nd. Edition, Krieger, Melbourne, FL, 1990.

49. [SHOO90b] Shooman, M. L., "Early Software Reliability Predictions," Software Reliability Newsletter, Technical Issues Contributions, IEEE Computer Society Committee on Software Engineering, Software Reliability Subcommittee, 9/11/90.

50. [SIEF89] Siefert, D. M., (March 1989), "Implementing Software Reliability Measures," *The NCR Journal*, Vol. 3, No. 1, pp. 24-34.

51. [STAR92] Stark, G. E., "Software Reliability Measurement for Flight Crew Training Simulators," *AIAA Journal of Aircraft*, Vol. 29, No. 3, May-June 1992, pp. 355-359.

52. [TAKA85] Takahashi, N. and Kamayachi, Y., "An Empirical Study of a Model for Program Error Prediction," *Proceedings 8th International Conference on Software Engineering*, London, pp. 330- 336.

53. [YAMA83] Yamada, S., Ohba, M., and Osaki, S., "S-Shaped Reliability Growth Modeling for Software Error Detection," *IEEE Transactions on Reliability*, Vol. R-32, No. 5, pp. 475-